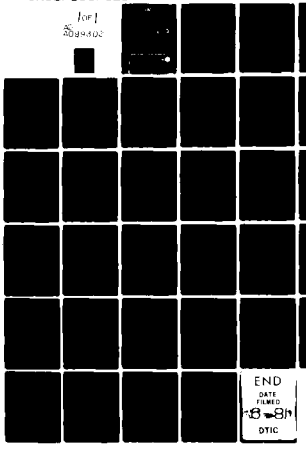
AD-A099 302 GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION A--ETC F/G 9/2
COMPLEXITY OF COMMUNICATION AMONG ASYNCHRONOUS PARALLEL PROCESS--ETC(U)
JAN 81 J E BURNS N00014-79-C-0873
UNCLASSIFIED GIT-ICS-81/01 NL



LEVEL II

AD _____

13

TECHNICAL REPORT
GIT–ICS–81/01

COMPLEXITY OF COMMUNICATION AMONG ASYNCHRONOUS PARALLEL PROCESSES

January, 1981

By

James E. Burns

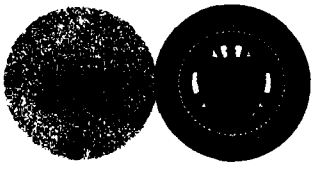
Prepared for

OFFICE OF NAVAL RESEARCH
800 N. QUINCY STREET
ARLINGTON, VA. 22217

Under

Contract No. N00014–79–C–0873
GIT Project No. G36–643

GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMPUTER SCIENCE
ATLANTA, GEORGIA 30332



THE RESEARCH PROGRAM IN
FULLY DISTRIBUTED PROCESSING SYSTEMS

COMPLEXITY OF COMMUNICATION AMONG ASYNCHRONOUS PARALLEL PROCESSES

TECHNICAL REPORT

GIT-ICS-81/01

James E. Burns

January, 1981

Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217

Contract Number N00014-79-C-0873
GIT Project Number G36-643

The Georgia Tech Research Program in
Fully Distributed Processing Systems
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

THE VIEW, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE NAVY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: GIT-ICS-81/01 — 2. GOVT ACCESSION NO.: AD-A099 302	3. RECIPIENT'S CATALOG NUMBER:
4. TITLE *(and Subtitle)*: Complexity of Communication Among Asynchronous Parallel Processes	5. TYPE OF REPORT & PERIOD COVERED: Technical Report, 1 Sep 79 - 14 Jan 81
	6. PERFORMING ORG. REPORT NUMBER: GIT-ICS-81/01
7. AUTHOR(s): James E. Burns	8. CONTRACT OR GRANT NUMBER(s): N00014-79-C-0873, NSF-[illegible]
9. PERFORMING ORGANIZATION NAME AND ADDRESS: School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research (ONR), 800 N. Quincy St., Arlington, VA 22217	12. REPORT DATE: Jan[illegible]81
	13. NUMBER OF PAGES: 130 + vii
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: same as item 11	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

same

18. SUPPLEMENTARY NOTES: Support was also provided by a Presidential Fellowship from the Georgia Institute of Technology and NSF grants MCS77-15628 and MCS77-28305. The view, opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Navy position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Asynchronous Systems
Mutual Exclusion
Deadlock
Shared Variables
Message Passing Systems

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

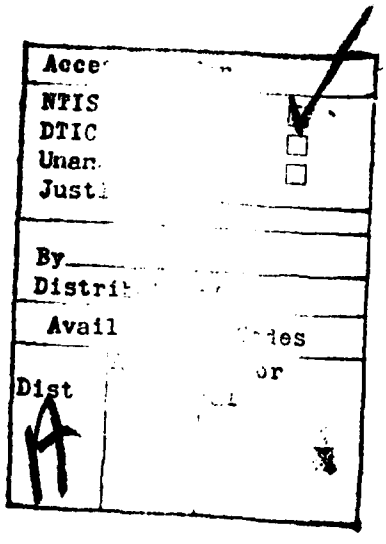
Certain problems of synchronization for systems of processes which execute asynchronously and communicate through shared variables or message passing are explored. Solutions are obtained for deadlock free mutual exclusion and lockout-free mutual exclusion for N processes communicating by shared variables. For systems which communicate by passing messages, a solution to the 'election problem' is presented - choosing a single process to become the system controller in an initial configuration of N →

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

processes in which no process has any information about the number or the identity of the other processes in the system.



SUMMARY

The thesis explores certain problems of synchronization for systems of processes which execute asynchronously. In a system of asynchronous parallel processes, the speed of execution of each process is independent of that of all the other processes in the system and may vary as the process executes. Two forms of communication are addressed in the thesis: communication through shared variables and communication by passing messages. The next three paragraphs discuss results using the shared variable model developed in the thesis. The final paragraph summarizes results about message passing systems.

A system satisfies mutual exclusion if it is impossible for two processes to simultaneously reach portions of their code called "critical sections". A system is deadlock-free if there is no computation in which every process continues to execute, but no process makes any progress. It is shown that N binary shared variables are necessary and sufficient to solve the problem of deadlock-free mutual exclusion for N processes which communicate only by atomic reads and writes of shared variables.

A system is lockout-free if there is no computation in which every process continues to execute, but some process does not make progress. Using a more powerful operation on shared variables (the generalized test-and-set), it is shown that lockout-free mutual exclusion can be solved for N processes with a single shared variable which takes on at most $\lfloor N/2 \rfloor + 9$ values.

A system satisfies n-exclusion (a generalization of mutual exclusion) if no more than n processes may simultaneously reach their critical sections. It is shown that the shared variables must be capable of taking on at least n(N-n) distinct values for a system of N processes which satisfies n-exclusion and has the property that there is a bound on how long a process must wait before it reaches its critical section.

The problem examined for systems which communicate by passing messages is called the "election problem". Beginning with an initial configuration in which no process has any information about the number or identity of the other processes in the system, a single process is chosen to become the system controller. It is shown that this can be solved by sending at most $4N + 6N \log N$ messages for a system of N processes connected in a ring network. It is also shown that more than $(1/8)N \log N$ messages must be sent in the worst case for such a ring of processes.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER I

INTRODUCTION

A system of parallel processes consists of many processing units which execute concurrently and can communicate with one another. In this thesis, we will examine systems in which the parallel processes are asynchronous. In an asynchronous system of parallel processes, the speed of execution of each process is independent of that of all the other processes in the system and may also vary as the process executes. Formal systems of asynchronous processes can be used to model multi-processing systems (in which processes communicate through shared memory) and distributed systems (in which processes communicate by sending messages over some communication medium). This thesis addresses the complexity of communication in such systems.

In systems with only one process (sequential systems), a problem specification often defines a function to be computed. The size of a given instance of a problem is measured by a function of the input. The complexity of a solution is measured by the amount of time or space used in computing the output value from the input value relative to the size of the input [AHU74]. In asynchronous systems,

there are problems which do not involve input and output. Instead, a solution is required to have a certain behavior for all executions (i.e., for all ways of interleaving the steps of the processes in the system). A problem for an asynchronous system is specified by a set of properties which must be satisfied. The size of a problem instance is measured in this thesis by the number of processes in the system. Therefore, a solution to such a problem must provide an algorithm for a system of N processes, where N is a positive integer. The complexity of a solution may be measured by some characteristic of the specified systems as a function of the number of processes in the system. The two measures used in this thesis are the size of the shared variable (when communication is through shared variables) and the number of messages sent (when communication is via messages).

The earliest paper dealing with asynchronous systems of the type studied here is by Dijkstra [Dij65]. Dijkstra considers the problem of synchronizing exclusive access to a single shared resource (mutual exclusion) by a system of asynchronous processes which can communicate only by reading or writing shared variables. Dijkstra's solution to this problem also has the property that the system cannot become deadlocked -- that is whenever some processes are competing for the resource, some process must succeed

within a finite number of steps. Knuth [Knu66] observes that Dijkstra's solution may allow some process to wait forever for the resource, even though other processes are being served. Knuth gives a solution to the mutual exclusion problem which guarantees that no process will wait forever for the resource; this new property is called "no-lockout". Knuth's solution actually satisfies a stronger property called "bounded waiting". A system satisfies the bounded waiting property if there is a bound on the number of times that another process may access and release the resource while the first process is waiting. Improvements to the waiting time in Knuth's algorithm are given by de Bruijn [deB67] and Eisenberg and McGuire [EiMc72]. Other variations of the mutual exclusion problem are studied by Lamport [Lam74], Rivest and Pratt [RP76], Peterson and Fischer [PF79] and Katseff [Kat78].

Cremers and Hibbard [CH78] first studied the complexity of the mutual exclusion problem with respect to the size of the shared variables. This work is extended by Burns, et. al. [BFJLP78], Peterson [Pet79b], and Fischer, et. al. [FLBB79]. Chapters II through V examine mutual exclusion problems with communication via shared variables.

In a 1977 paper [LeL77], Le Lann proposes a synchronization problem, called the "election problem", for an asynchronous system in which the processes, connected in

a ring, communicate by sending messages. The processes must collectively choose one of their number to be "elected" to take control of the system. Chang and Roberts [CR77] and Hirschberg and Sinclair [HS79] provide improvements over Le Lann's algorithm in terms of the number of messages sent. Chapter VI gives an improved algorithm and a lower bound.

Chapter II defines a formal model for asynchronous systems. The model used is based on the models of Lipton [Lip73], Burns, et. al. [BFJLP78], and Lynch and Fischer [LF79]. The model is used to specify problems and to prove results in Chapters III, IV and V.

Chapter III examines Dijstra's original problem, deadlock-free mutual exclusion. Processes are allowed to communicate only by atomically reading or writing shared variables. It is shown that N binary variables are necessary and sufficient to solve this problem for an asynchronous system of N processes.

Chapter IV examines the problems of lockout-free mutual exclusion and bounded-waiting mutual exclusion without any constraint on the way shared variables are accessed. Processes are allowed to execute "test-and-sets" on shared variables. A test-and-set may read a variable and re-write it as a function of the value read, all in a single, indivisible operation. The main results of the

chapter are algorithms which solve lockout-free mutual exclusion for N processes with a ⌊N/2⌋+9-valued shared variable and solve bounded-waiting mutual exclusion with an N+5-valued shared variable. These results have appeared as part of joint work with M.J. Fischer, P. Jackson, N.A. Lynch, and G.L. Peterson [BFJLP78].

Chapter V examines another problem of synchronization. Instead of a single resource, many identical resources are to be shared among a set of asynchronous processes. Each process must have exclusive access to a resource while using it and may only use one resource at a time. The main result of the chapter is that a shared variable which can take on at least n(N-n) values is needed to share n resources amoung N processes in a deadlock-free system. The results in this chapter have appeared as part of joint work with M.J. Fischer, N.A. Lynch, and A. Borodin [FLBB79].

Chapter VI examines the election problem. An improved algorithm which will send no more than 4N + 6N log N messages for a ring of N processes is presented. It is also shown that any solution to the problem must send more than (1/8)N log N messages in the worst case.

The final chapter summarizes the work in the thesis and indicates areas for future research.

CHAPTER II

ASYNCHRONOUS SYSTEMS

Reasoning about parallel systems is a difficult task. The English language, and the terminology that has been developed for describing sequential systems, can lead to ambiguities and misunderstandings when used to discuss systems with concurrent activities. For example, consider the description (paraphrased below) of the problem of the Five Dining Philosophers given by Dijkstra [Dij71]. Five philosophers sit at a round table. A single fork lies between each adjacent pair of philosophers. Each philosopher alternately thinks and eats. In order to eat, a philosopher must first obtain the forks which are on either side of him. Dijkstra presents a somewhat complex solution which has one globally used semaphore [Dij68a], five shared variables and five private semaphores. (Note: Dijkstra's paper illustrates the development of a correct solution and is not concerned with optimality at all.) The major constraint of the problem is that no two adjacent philosophers may be eating simultaneously. In order to rule out the straightforward solution of using a single global semaphore (allowing only one philosopher to eat at a time), a condition which requires that two non-adjacent

philosophers be always "allowed to eat" at the same time if their neighbors are not eating is needed. The following solution meets this added constraint, and requires only N private semaphores for N philosophers. (Note: this solution is probably obvious, but it has not appeared previously to my knowledge.)

Program for Philosopher w, where w is odd

```
cycle begin think;
           P(fork[w]);
           P(fork[(w+1) mod N]);
           eat;
           V(fork[w]);
           V(fork[(w+1) mod N]);
      end;
```

Program for Philosopher w, where w is even

```
cycle begin think;
           P(fork[(w+1) mod N]);
           P(fork[w]);
           eat;
           V(fork[(w+1) mod N]);
           V(fork[w]);
      end;
```

A Solution to the Problem of the Dining Philosophers

Figure 2-1

Number the philosophers from 0 to N-1 clockwise around the table, where N is the number of philosophers, N>1. The philosophers with odd numbers attempt to pick up their left fork first, while those with even numbers pick

up their right fork first. (The program, using Dijkstra's style, is given in Figure 2-1. The semaphore array fork[0:N-1] is initialized to 1.) It is easy to see that deadlock cannot occur. Since two adjacent philosophers cannot be blocked waiting for the fork (semaphore) between them, deadlock can only occur if every philosopher holds exactly one fork. But this is impossible because philosophers 0 and 1 both pick up fork[1] first and so cannot both hold exactly one fork.

This solution (apparently an unintended one) could be ruled out by adding a constraint that all philosophers must behave in the same way (have identical programs), but, in Dijkstra's informal description of the problem, this requirement is not mentioned. We will try to avoid misunderstandings by using formal definitions. Of course, formality is no guarantee that an error or omission will not be made in defining a problem, but at least there should be a smaller chance of the reader having a different interpretation of the problem than was intended. A formal model for asynchronous systems is developed in this thesis to allow precise problem descriptions. This will enable us to prove that algorithms satisfiy their specifications and to prove lower bound results.

The model presented here is not innovative, but rather draws heavily on the foundation laid by Lipton

[Lip73], Burns, et al. [BFJLP78], Lynch and Fischer [LF79], and others. The objective is to define a model which is well suited to the purposes of this thesis. Therefore, the model is tailored to the task of specifying problems of synchronization and to the task of analyzing solutions to such problems.

Notation

Let A and B be sets. The union of A and B is A U B, and the product of A and B is AxB. If a is an element of A, then a ∈ A. If f is a function from A to B, then f: A -> B. The number of elements in A is |A|. If n is a positive integer then [n] is the set {1,...,n}.

Let h and h' be sequences. The concatenation of h and h' is denoted hh'. The null sequence is the sequence with length zero. If every element of h is in a set A, then h is a sequence over A. The term n-tuple is often used for a finite sequence of length n. If h is an n-tuple and i ∈ [n], then h.i is the i^{th} element of h.

Asynchronous Systems

An asynchronous system (defined formally below as an "(M,N)-system") consists of a set of N independent processes and M shared variables. All communication takes place through the shared variables. Each time a step occurs in an asynchronous system, a non-deterministic

choice is made to decide which process will execute the step. However, once this choice is made, the outcome is fully determined. That is, each process appears to be deteministic when examined in isolation; all non-determinism in the system comes from the interleaving the order in which the processes execute.

Formally, an (M,N)-system is a 4-tuple, $S = (V,X,p,q_0)$, where $V = V_1xV_2x...xV_M$ (V_j is the set of values of the j^{th} variable), $X = X_1xX_2x...X_N$ (X_i is the set of states of the i^{th} process), p is an N-tuple of transition functions from VxX to VxX and q_0 is a distinguished element of VxX called the initial instantaneous description. The i^{th} component of p, p.i, is the transition function of process i. "Process i" is often abbreviated by "Pi". An instantaneous description (id) is an element of VxX. For any id, q = (v,x), define V(q) = v, Vj(q) = v.j for j in [M], X(q) = x, and Xi(q) = x.i for i in [N].

The transition function of process i is a total function, $p.i: VxX_i -> VxX_i$. Let total function $p_i: VxX -> VxX$ be the extension of p.i such that for every id q = (v,x) S, $p_i(v,x) = (v',x')$ if and only if $p.i(v,x.i) = (v',x'.i)$ and x'.j=x.j for j≠i. That is, p_i is the extension of p.i to ids in the natural way. When a process executes a step, it cannot see or change the

state of any other process. An id q *looks like* an id q' to a set of processes if V(q) = V(q') and if Xi(q) = Xi(q') for every Pi in the set.

If q and q' are ids of S such that $p_i(q) = q'$, then write $q \xrightarrow{i} q'$. If $q \xrightarrow{i} q'$ for some $i \in [N]$, then $q \rightarrow q'$. If $q_1, q_2, \ldots, q_k$ are ids of S such that $q_i \rightarrow q_{i+1}$ for $1 \leq i < k$, then q_k is *reachable* from q_1. (That is, reachability is the reflexive transitive closure of the "->" relation.)

Schedules

A *schedule* of an (M,N)-system S is any finite of infinite sequence over [N]. The result function, *r*, of S is defined for any id q of S and any finite schedule h of S so that if h is the null sequence then r(q,h) = q and if h = h'i, where $i \in [N]$, then $r(q,h) = p_i(r(q,h'))$. The result function, r(q,h), gives the resulting id when S is started in id q and the processes of S take steps in the order specified by h.

Schedules are used to specify computations of S in order to reason about the behavior of the system. Let q_1 be an id of S and $h = i_1 i_2 \ldots$ be any schedule (finite of infinite) of S. The computation of h applied to q_1 is $comp(q_1,h) = q_1 i_1 q_2 i_2 q_3 \ldots$, where $q_m \xrightarrow{i_m} q_{m+1}$ for $m \leq 1$. A transition $q \xrightarrow{i} q'$ *occurs in* $comp(q_1,h)$ if there is an integer j such that $q = q_j$, $i = i_j$ and $q' = q_{j+1}$. The *id sequence* of $comp(q_1,h)$ is $q_1 q_2 \ldots$

Critical Systems

The problems examined in Chapters III, IV and V all involve mutual exclusion in asynchronous systems. In this type of problem, a process is assumed to have a certain section of its program which is "critical". A critical section is intended to represent that part of a program which might affect or be affected by the action of another process. For example, if two processes concurrently update the same disk file, errors may be introduced; the segments of code doing the update could be considered critical sections. It is often useful to synchronize the execution of critical sections in an asynchronous system. The following definitions formalize systems with critical sections.

An (M,N)-system, $S=(V,X,p,q_0)$, is *critical* if for every $i \in [N]$ there is a partition of X_i into sets R_i, T_i, C_i and E_i such that the following conditions hold for every id q of S.

If $Xi(q) \in R_i \cup T_i$, then $Xi(p_i(q)) \in T_i \cup C_i$ (2.1)

If $Xi(q) \in C_i \cup E_i$, then $Xi(p_i(q)) \in E_i \cup R_i$ (2.2)

The sets R_i, T_i, C_i and E_i are referred to as the *remainder*, *trying*, *critical* and *exit regions* of process i, respectively. Equation 2.1 implies that a process in its remainder region will always leave the remainder region and

go to either the trying region or critical region on its next step. A process may stay in its trying region for any number of steps, but it must then go to the critical region. Equation 2.2 has symmetric implications for a process in the critical region or the exit region. Note: the partitions of the X_i are assumed to be fixed for any critical system under discussion.

The intended interpretation is that the critical region corresponds to the critical section of a process, while the remainder region corresponds to the parts of a process which do not contain critical sections. A process is not supposed to communicate with others while in either of these regions, so the formal definition suppresses all detail within the critical and remainder regions. All communication among the processes in a critical system occurs within the trying and exits regions, which contain the required synchronization protocols.

Fair Mutual Exclusion

The mutual exclusion problem for asynchronous systems was first studied by Dijkstra [Dij65]. Later authors [Knu66, EiMc72, RP76] added new fairness constraints and provided algorithms to meet these constraints. The following definitions formalize these early concepts in a way very close to that given in Burns, et al. [BFJLP78].

A critical (M,N)-system, $S = (V,X,p,q_0)$, satisfies mutual exclusion if for every id q of S which is reachable from q_0 there is at most one $i \in [N]$ such that $Xi(q) \in C_i$. That is, S satisfies mutual exclusion if two processes cannot reach their critical regions at the same time when S is started in its initial id.

Mutual exclusion is a property which involves only finite schedules. The next two properties apply only to infinite computations. Because of the assumption that processes have non-zero speed (that is, steps of the processes continue to occur in the computation), we will only be interested in a certain subset of all possible infinite computations. A process, Pi, is said to halt in schedule h if i occurs only a finite number of times in h. For any id q of S, a schedule, h, of S is R-admissible from q if for every pair of schedules, h' and h", of S such that h' is finite and $h = h'h"$, and for every $i \in [N]$, either $Xi(r(q,h')) \in R_i$ or i occurs in h". That is, a schedule h is R-admissible from q if every process which halts in h ends up in its remainder region in comp(q,h).

Process i changes regions in comp(q,h) if there exist finite prefixes h' and h" of h such that Xi(r(q,h')) is in a different region of X_i from Xi(r(q,h")). A region change occurs in comp(q,h) if some process changes regions in comp(q,h).

A infinite schedule h exhibits deadlock from an id q if there is a non-null tail h" of h such that h = h'h" and no region change occurs in comp(r(q,h'),h"). A critical system S is deadlock-free (satisfies "no deadlock") if no schedule h of S which is R-admissible from q_0 exhibits *deadlock from* q_0. *The deadlock free property is used to* prohibit trivial solutions to the mutual exclusion problem. Note that the definition of deadlock given here differs from that sometimes used where a system is called "deadlocked" at id q only if every schedule exhibits deadlock from q.

An infinite schedule h locks out process i from id q if there is a non-null tail h" of h such that h = h'h", $Xi(r(q,h')) \notin R_i$ and Pi does not change regions in *comp(r(q,h'),h"). S is lockout-free (satisfies "no* lockout") if for every i ∈ [N], no schedule h of S which is R-admissible from q_0 locks out Pi from q_0. That is, in a lockout-free system any process which is not in its remainder region will change regions after a finite number of system transitions in any R-admissible computation.

The lockout-free property guarantees that a waiting process will eventually be served. A stronger fairness condition is provided by the bounded waiting property *(given below).*

Process i cycles b times in comp(q,h) if there exist *finites prefixes* $h_1, h_2, \ldots, h_{2b}$ of h such that for $j \in [2b-1]$, h_j is a proper prefix of h_{j+1}, *and for* k ∈ [b], $Xi(r(q,h_{2k-1})) \in R_i$ and $Xi(r(q,h_{2k})) \in C_i$. Process i b-waits in comp(q,h) if Pi is not in its remainder region at q, Pi does not change regions in *comp(q,h) and some process cycles b times in comp(q,h). S* satisfies b-bounded waiting if for every id q reachable from q_0 and every schedule h of S, no process of S (b+1)-waits in comp(q,h). Thus, in a system which satisfies b-bounded waiting, a process waiting for service can be passed at most b times by any other process.

CHAPTER III

MUTUAL EXCLUSION USING READS AND WRITES

In the original specification of the mutual exclusion problem [Dij65], all communication was required to be through shared variables using only indivisible reads and writes. Dijkstra's original solution, and those of Knuth [Knu66], de Bruijn [deB67] and Eisenberg and McGuire [EiMc72] all use $N*3^N$ shared states for N processes. This chapter will show that 2*N shared states are necessary and sufficient to solve the problem of deadlock-free mutual exclusion using only indivisible reads and writes for accessing shared variables.

Read/write Systems

Let $S = (V,X,p,q_0)$ be an (M,N)-system, $j \in [M]$ and $i \in [N]$. Process i is ready to read variable j at q if and only if for every id q' if $Xi(q') = Xi(q)$ then $V(p_i(q')) = V(q)$, and if in addition $Vj(q') = Vj(q)$ then $Xi(p_i(q')) = Xi(p_i(q))$. That is, Pi will not change the value of any shared variable, and the next state of Pi depends only on the value of V_j. If Pi is ready to read variable j at q and $p_i(q) = q'$, then $q \xrightarrow{i} q'$ is a read of variable j by Pi. (Note that $q \xrightarrow{i} q'$ may technically

qualify as a read of more than one variable if Pi actually looks at no variables.)

Let $w \in V_j$. Process i is ready to write variable j with value w at q if and only if for every id q' if $Xi(q') = Xi(q)$ then $Xi(p_i(q')) = Xi(p_i(q))$, $Vj(p_i(q')) = w$, and for every $k \in [M]$, $k \neq j$ implies that $Vk(p_i(q')) = Vk(q')$. That is, the next state of Pi is independent of the value of V, Pi writes a fixed value into variable j, and Pi does not affect the value of any other variable. If Pi is ready to write variable j at q with value w and $p_i(q) = q'$, then $q \xrightarrow{i} q'$ is a write of variable j with value w by Pi.

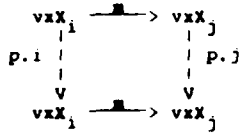
An (M,N)-system $S = (V,X,p,q_0)$ has the read/write property if for every $i \in [N]$ and every id q of S, Pi is either ready to read or ready to write a variable at q.

Symmetry

Dijkstra included the following constraint in his definition of the mutual exclusion problem: "The solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority." [Dij65] This intuitive property seems difficult to formalize, and has been dropped by some authors (Lamport [Lam74], Rivest and Pratt [RP76] and Peterson and Fischer [PF77], for example). The strongest definition of symmetry (identically programmed) does not allow a solution to the

mutual exclusion problem, as shown below.

Let $S = (V,X,p,q_0)$ be a critical read/write (M,N)-system. The processes of S are <u>identically programmed</u> if for every $i,j \in [N]$ there is a bijection $m: X_i \rightarrow X_j$ which preserves the regions of X_i such that $m(Xi(q_0)) = Xj(q_0)$ and such that if $x,x' \in X$ and $m(x.i) = x'.j$ then for all $v \in V$, $V(p_i(v,x)) = V(p_j(v,x'))$ and $m(X_i(p_i(v,x)) = X_j(p_j(v,x'))$; i.e., for any $v \in V$ the following diagram commutes.



That is, there is an isomorphism between the states of every pair of processes in S, and all processes start in isomorphic states.

<u>Theorem 3.1</u>

There is no critical read/write (M,N)-system (N>1) with identically programmed processes which is deadlock-free and satisfies mutual exclusion.

<u>Proof</u>: Suppose $S = (V,X,p,q_0)$ is such a system. Consider the schedule h = 123...N and any id q of S for which the states of all the processes are isomorphic to one another. All the processes will still be isomorphic to one another at r(q,h) because either they all do reads, and read the same value, or they all write the same value. Thus, for any schedule h' = hh...h, all processes of S are in isomorphic states at $r(q_0,h')$. But then if any process is critical at $r(q_0,h')$, then all must be critical, which would violate mutual exclusion. Let h* = hh... , the concatenation of h with itself an infinite number of times. No process can ever reach its critical region in $comp(q_0,h^*)$. Since a process must reach its critical region after at most three region changes, schedule h* exhibits deadlock from q_0. Since h* must be R-admissible from q_0, S cannot be deadlock-free. S cannot satisfy both mutual exclusion and no deadlock, so the theorem is proved. □

Since the "identically programmed" notion of symmetry is inconsistent with the requirements of the mutual exclusion problem for read/write systems, it will not be considered further here. While it may be possible to formulate formal definitions of symmetry which agree with our intuitions and still allow solutions of the mutual exclusion problem, such definitions will not be sought in this thesis. For the remainder, we allow asymmetric solutions.

Deadlock-free Mutual Exclusion

An algorithm which solves the problem of deadlock-free mutual exclusion using only indivisible reads and writes is given in Figure 3-1. The figure gives the program for each process i, i ∈ [N]. At the initial id of the system, all processes are at the beginning of their programs, and the shared variables all have value "down".

```
program Process_i;
  type flag = (down,up);
  shared var F : array [1..N] of flag;
  var j : 1..N;
begin
  while true do begin
  1: remainder;      (* remainder region *)
  2: F[i] := down;   (* begin trying region *)
  3: for j := 1 to i-1 do
     if F[j] = up then goto 3;
  4: F[i] := up;
  5: for j := 1 to i-1 do
        if F[j] = up then goto 3;
  6: for j := i+1 to N do
        if F[j] = up then goto 7;
  7: critical;       (* critical region *)
  8: F[i] := down;   (* exit region *)
     end
  end.
```

Deadlock-free Mutual Exclusion

Figure 3-1

Theorem 3.2

For every N>0 there exists a critical read/write (N,N)-system which solves the problem of deadlock-free

mutual exclusion and for which $|V_i| = 2$ for all $i \in [N]$.

Proof: For any N>0, let $S = (V,X,p,q_0)$ be the system in which process i has the program given in Figure 3-1, $Vi(q_0)$ = down and $Xi(q_0)$ = the statement labeled 1 in the program in Figure 3-1, for each $i \in [N]$. Labels 1, 7 and 8 denote the remainder, critical and exit regions, respectively. The trying regions corresponds to statements 2 through 6. Clearly, S is a critical read/write (N,N)-system.

Suppose deadlock can occur. Then there is an id q reachable from q_0 and a schedule h which is R-admissible from q such that some process is not in remainder at q and yet no process changes regions in comp(q,h). Since the only backward branches in each process's program occur in the trying region, observe that for each $i \in [N]$, either $Xi(q) \in R_i$ and i does not occur in h or $Xi(q) \in T_i$ and i occurs infinitely often in h. The set of processes which are not in remainder at q are called "active".

For each $i \in [N]$, define the following subsets of T_i. A_i = the sets of states of Pi corresponding to the statements labeled 2 and 3. B_i = the sets of states of Pi corresponding to the statement labeled 6. Note that if Pi reaches B_i, then it will remain there for the rest of the computation and F[i] will be continuously equal to "up". Let m = min {i ∈ [N] : Pi is active at q}. Since m will

eventually detect that no F[i] = up for i ∈ [m-1], Pm will reach B_m after a finite prefix, h_1, of h (let h = h_1h_2). (That is, $X_m(q') \in B_m$, where $q' = r(q,h_1)$). After some finite prefix, h_3, of h_2 (let $h_2 = h_3h_4$), every active Pi will either be in B_i or will begin cycling forever in A_i with F[i] = down, since all active processes which do not reach B_i will detect F[m]=up. Let n = max {i ∈ [N] : $Xi(q") \in B_i$}, where $q"=r(q',h_3)$. Now Pn will find all F[i]=down for i ∈ {n+1,...,N}, so Pn will change regions in comp(q",h_4), contradicting the supposition. Therefore, deadlock cannot occur.

Now suppose that mutual exclusion may be violated. Then there must be values i,j ∈ [N] such that i≠j and a finite schedule h such that $q=r(q_0,h)$ and $Xi(q) \in C_i$ and $Xj(q) \in C_j$. Let D_i = the set of states of Pi corresponding to statements 5, 6 and 7 of the algorithm, and D_j be similarly defined for Pj. Pi may enter and leave D_i several times before reaching its critical region, but there must be an id at which Pi enters D_i for the last time before going critical. Let q_a be this id for Pi, and let q_b be a similar id for Pj, in the id sequence $q_0q_1...q_k$ of comp(q_0,h) (where $q=q_k$). Assume without loss of generality that a<b. But then for every c, a≤c≤k, F[i]=up at q_c. Since Pj must test F[i] after entering D_j (either at statement 5 or 6), Pj cannot go

critical in the id sequence $q_bq_{b+1}...q_k$, contradicting the supposition. Therefore the algoritnm also satisfies mutual exclusion and the theorem is proved. □

A Corresponding Lower Bound

Some additional definitions are needed for the lemmas that lead up to the lower bound theorem. As before, let S be a critical read/write system. Let q_1 be an id of S, h be a schedule of S, i ∈ [N], m ∈ [M] and $q_1q_2...$ be the id sequence of comp(q_1,h). If there exist positive integers j<k such that $q_j \rightarrow q_{j+1}$ is a write of Vm by Pi and $q_k \rightarrow q_{k+1}$ is a write of Vm, and if for all n, j<n<k, $q_n \rightarrow q_{n+1}$ is not a read of Vm by any process other than Pi, then the write of Vm by Pi at q_j is <u>obliterated</u> in comp(q,h). A write which is obliterated cannot affect the state of any process except the writing process itself.

If Pi is ready to write variable j at id q and if $V(q) \neq V(p_i(q))$, then Pi is <u>ready to change</u> variable j at id q. If Pi is ready to change variable j at id q, then $q \rightarrow p_i(q)$ is a <u>change</u> of variable j. Thus, a change is a more restricted kind of write.

If i ∈ [N], q is an id of S, h_1 and h_2 are schedules of S (h_1 finite) such that $Xi(r(q,h_1)) \in R_i$ and such that every change by Pi is obliterated in comp($r(q,h_1),h_2$), then Pi is <u>hidden</u> in comp(q,h_1h_2).

That is, Pi is hidden in a computation if every change that it makes after last leaving its remainder region is overwritten. If Pi is hidden in a computation, then every other process in the computation must behave as if Pi is still in its remainder region, since they cannot detect that it has taken any steps.

Lemma 3.1

Let S be a critical read/write system, h be a finite schedule of S and Pi be a process of S which is hidden in $comp(q_0,h)$. If $q=r(q_0,h)$ then there is an id q' of S reachable from q_0 such that $Xi(q') \in R_i$, $V(q')=V(q)$ and $X_j(q')=X_j(q)$ for all $j \neq i$, $j \in [N]$ (that is, q' looks like q to all processes other than Pi).

Proof: Let h_1 be the longest prefix of h for which $Xi(r(q_0,h_1)) \in R_i$ (h_1 exists since Pi is hidden in $comp(q_0,h)$), and let $h = h_1h_2$. Let h_3 be the schedule obtained from h_2 by removing all occurences of i, and let $h' = h_1h_3$. Now $q'=r(q_0,h')$ meets the requirements of the lemma since Pi cannot have left the remainder region since $r(q_0,h_1)$. Note that $V(q')=V(q)$ because all changes by Pi in the computation from $r(q_0,h_1)$ to q are obliterated. Also, $X_j(q')=X_j(q)$ for $j \neq i$ because no process can have read anything changed by Pi since $r(q_0,h_1)$. □

Let S be a critical read/write system, q be an id of S, $i \in [N]$ and $j \in [M]$. If Pi is ready to write variable j at q, then variable j is covered at q by Pi. Since a covered variable can be overwritten at any time (with an appropriately chosen schedule) we can obliterate any writes which are made to these variables without any intervening reads. If every change that a process has made since leaving the remainder region is to a covered variable, then the process can be hidden.

Lemma 3.2

Let S be a critical read/write system with at least two processes which solves deadlock-free mutual exclusion, h be a finite schedule of S and Pi be a process of S hidden in $comp(q_0,h)$. If Pi goes critical on its own from $q=r(q_0,h)$ by a schedule $h_1 = ii...i$, then in $comp(q,h_1)$ Pi must change some variable which is not covered by any other process at q.

Proof: Suppose Pi goes critical from q by schedule h_1 without changing any variable which is not covered by some other process at q. Let h_2 a schedule consisting of exactly one step of each process other than Pi. Then every change of Pi is obliterated in $comp(q,h_1h_2)$, so Pi is hidden in $comp(q,h_1h_2)$. By Lemma 3.1, there is a reachable id q" which looks like $q'=r(q,h_1h_2)$ to all the other processes but has Pi in remainder. Since S is

deadlock-free, some other process Pj≠Pi of S can go critical from q" by schedule h' not containing i. But then r(q',h') has both Pi and Pj critical, contradicting mutual exclusion. □

In order to show that N shared variables are necessary for a critical read/write (M,N)-system to solve deadlock-free mutual exclusion, we want to show that (at some point) each process must have a variable for its exclusive use. As is frequently the case, we will prove a stronger lemma so that the induction will go through. The lemma shows that N variables can be covered by N hidden processes (we say that the variables are "nullified"). Let S be a critical read/write system, q be an id of S, h be a finite schedule of S and W be a subset of V. W is nullified in comp(q,h) if for every w ∈ W there is a process which is hidden in comp(q,h) and which is ready to change w at r(q,h).

Lemma 3.3

Let S be a critical read/write (M,N)-system with N≥2 processes which solves deadlock-free mutual exclusion, and let q be any reachable id of S at which all processes of S are in their remainder regions. For every K, 1≤K≤N, there is a finite schedule h of S using only processes P1,P2,...,PK (i.e., h is over [K]) such that K variables are nullified in comp(q,h).

Proof: The proof is by induction on K, the number of variables nullified.

BASIS. Let K=1. By no deadlock, there must be a finite schedule h' consisting only of 1's such that P1 goes critical at r(q,h'). By Lemma 3.2, there must be a prefix, h" of h' such that P1 is hidden (i.e., Pi has not changed any variables) in comp(q,h") and is ready to change some variable, w, at r(q,h"). But then {w} is nullified in comp(q,h") and the lemma holds for K=1.

INDUCTIVE STEP. Assume the lemma holds for K = k-1. By the inductive assumption, there is a finite schedule h_0 using only processes P1,...,P(k-1) such that a set, W_1, of k-1 variables is nullified in comp(q_0,h_0). Let q_1 = r(q_0,h_0). From q_1 successively find id's q_2,q_3,... by finite schedules h_1,h_2,... such that q_{i+1} = r(q_i,h_i), where h_i is defined in the following way. For each i>0, let h_i begin with the prefix 123...(k-1). From r(q_i,123...(k-1)), find an extension of h_i which returns P1,...,P(k-1) to their remainder regions (by no deadlock this extension exists). Finally, complete h_i by appending a schedule which nullifies a set, W_{i+1}, of k-1 variables at q_{i+1}. The final portion of h_i exists by the inductive assumption.

For each i>0, Lemma 3.2 implies that Pk can be moved on its own by some shortest schedule s_i from q_i such that

Pk is ready to change some variable, w_i, which is not in W_i. (Note that $comp(q_i,s_ih_i)$ hides Pk, and so does an{ extension of s_ih_i which does not included steps of Pk.) Since there are only N variables, there must be integers $0 < i < j < m < 2N+1$ such that $w_i = w_j = w_m$. If the value that Pk is ready to write at $q' = r(q_i,s_ih_ih_{i+1}...h_j)$ is the same that Pk is ready to write at $r(q_j,s_j)$, then Pk is ready to change w_i at q' and the lemma holds since $\{w_i\} \cup W_j$ is nullified by $comp(q_i,s_ih_i...h_j)$. Otherwise Pk is ready to write different values at q' and $r(q_j,s_j)$. In this case, Pk must be ready to change w_i either at $r(q',h_j...h_m)$ or at $r(q_j,s_jh_j...h_m)$. This implies that $\{w_i\} \cup W_m$ is nullified either by $comp(q_i,s_ih_i...h_m)$ or by $comp(q_j,s_jh_j...h_m)$, and the lemma is proved. □

Theorem 3.3

If $S=(V,X,p,q_0)$ is a critical read/write (M,N)-system with at least two processes and S solves deadlock-free mutual exclusion, then S must have at least N variables and $|V| \geq 2^N$.

Proof: By no deadlock, there is an id q reachable from q_0 such that all processes of S are in their remainder regions at q. Apply Lemma 3.3 for q to find another reachable id, q', which nullifies N distinct variables. Since we can choose the order in which the

processes execute from q', we change any subset of the N variables to new values. This implies that $|V| \geq 2^N$. □

CHAPTER IV

LOCKOUT-FREE MUTUAL EXCLUSION

The model described in Chapter II allows a very powerful form of access to shared variables. A variable can be read and re-written as a function of the value read, all in a single indivisible operation. This primitive operation has been called a "generalized test-and-set" [BFJLP78]. Cremers and Hibbard showed that three shared values are necessary and sufficient to solve the problem of lockout-free mutual exclusion for two processes, even when using a generalized test-and-set [CH78]. This chapter extends their result to N processes, for N≥1. These algorithm given below (Algorithm A) was developed as a collaborative effort with my co-authors: Michael J. Fischer, Paul Jackson, Nancy A. Lynch and Gary L. Peterson [BFJLP78]. (The reader is urged to consult Peterson [Pet79b, Pet80] for further refinements.) However, the main contribution of this chapter is in the proof of correctness of the alorithm, which is new.

Lower Bounds

This section states lower bound results on the number of shared values for systems satisfying lockout-free mutual

exclusion. Note that these lower bounds trivially apply to all systems which use communication primitives (such as indivisible reads and writes) which are more restrictive than the generalized test-and-set.

Theorem 4.1 (Burns, Fischer, Lynch, Jackson and Peterson)

If $S=(V,X,p,q_0)$ is a critical (M,N)-system, N≥1, that satisfies lockout-free mutual exclusion, then $|V| \geq SQRT(2N) - 1/2$.

This result may be strengthened if a restriction is placed on how much information a process can "remember" while it is in its remainder region. A critical (M,N)-system, $S = (V,X,p,q_0)$, is memoryless if for all $i \in [N]$, $|R_i| = 1$. All mutual exclusion algorithms known to the author can be modified to be memoryless without increasing the amount of shared memory, so this restriction may not be as severe as it seems.

Theorem 4.2 (Burns, Fischer, Lynch, Jackson and Peterson)

If $S=(V,X,p,q_0)$ is a critical (M,N)-system, N≥1, that satisfies lockout-free mutual exclusion and is memoryless, then $|V| \geq N/2$.

This chapter will show that the bound of Theorem 4.2 is tight to within an additive constant.

Notation for the Generalized Test-and-Set

It is important that a reader be able to translate

algorithms into the formal model without any ambiguity. The notation explained here precisely defines the indivisible accesses (test-and-sets) to a shared variable.

```
<test-and-set> ::= test <shared variable> until
                       <set-list> endtest

<set-list>     ::= <set>  |  <set> ; <set-list>

<set>          ::= <value_1> setto <value_2>  |
                   <value_1> setto <value_2> : <statement>
```

Test-and-set Syntax

Figure 4-1

The intended semantics for <test-and-set> is to compare the value of the <shared variable> with the $<value_1>$ values in the <set-list>. If a match is found, <shared variable> is set to the corresponding $<value_2>$, the associated <statement> (if any) is executed and control is passed to the statement following the <test-and-set>. If no match is found, the <shared variable> is unchanged and control returns to the beginning of the test-and-set. Note that each iteration of the implied loop is a single, indivisible operation, so other processes may access the variable between iterations. The implied loop is included because this simplifies the writing of the program given


```
program A;      (* for N processes *)
  const k = ⌊N/2⌋;
  type msg = (S0,S1,...,Sk,FREE,STOP,
              ENTER,AE,GO,AG,ELECT,QUERY,ONE,BYE);
  shared var V: msg initial FREE;
  var  M: msg initial S0;
       main, buff, idlers: integer initial 0;
begin
 REMAINDER:                          (* remainder region *)

 TRYING:                             (* enter trying region *)
T0: test V until
      FREE  setto S0  : goto CRITICAL;
      Sj    setto Sj+1: goto BUFFER;   (* 0≤j<k *)
      Sk    setto S0  : goto EXECUTIVE;
      STOP  setto S1  : goto IDLE;
      other setto S1  : (M := other;    (* other: ENTER,AE,GO,AG, *)
                         goto HOLDING)  (* ELECT,QUERY,ONE,BYE    *)
    endtest;

 EXECUTIVE:
    buff := k;
    while buff > 0 do
T1:   test V until
        Sj    setto STOP: (buff := buff + j - 1;  (* 0≤j≤k *)
                           idlers := idlers +1);
        other setto S0  : M := other    (* other: FREE,ENTER,AE,GO*)
      endtest;                          (* AG,ELECT,QUERY,ONE,BYE *)
    if M=FREE then goto CRITICAL;
    If M ≠ S0 then
T2:   test V until S0 setto M: M := S0  endtest;
T3: test V until
      FREE  setto S0  : goto CRITICAL;
      Sj    setto Sj+1: goto BUFFER;        (* 0≤j≤k *)
      other setto S1  : M := other      (* other: ENTER,AE,GO,AG, *)
    endtest;                            (* ELECT,QUERY,ONE,BYE    *)

 HOLDING:
T4: test V until
      S0    setto M: (M := S0; goto BUFFER);
      STOP  setto M: (M := S0; goto IDLE)
    endtest;

 BUFFER:
T5: test V until
      ENTER setto AE: goto MAINAREA;
      STOP  setto S0: goto IDLE
    endtest;

 IDLE:
T6: test V until GO setto AG: goto MAINAREA  endtest;

 MAINAREA:
T7: test V until ELECT setto S0  endtest;

 CONTROL:                            (* become controller *)
    while true do begin
T8:     test V until
          Sj setto QUERY: buff := buff + j    (* 0≤j≤k *)
        endtest;
        while true do
T9:       test V until
            Sj  setto Sj: buff := buff + j;   (* 0≤j≤k *)
            ONE setto S0: (main := main + 1;
                           exit);
            BYE setto S0: goto CRITICAL
          endtest;
    end;
```

Program for Lockout-free Mutual Exclusion

Figure 4-2

```
   CRITICAL:                              (* critical region *)

   EXIT:                                  (* enter exit region *)
   while idlers > 0 do begin              (* awaken idlers *)
E0: test V until
      Sj setto S0: buff := buff + j       (* 0<j<k *)
    endtest;
    idlers := idlers - 1;
    main   := main   + 1;
    while true do
E1:   test V until
        Sj setto Sc: buff := buff + 1;    (* 0<j<k *)
        AG setto S0: exit
      endtest
    end;

    if (main=0 and buff=0) then
E2:   test V until
        S0 setto FREE: goto REMAINDER;
        Sj setto S0  : buff := buff + j   (* 0<j<k *)
      endtest;

    if main=0 then          (* move processes: buffer to main *)
      while buff > 0 do begin
E3:     test V until
          Sj setto ENTER: (buff := buff + j - 1;  (* 0<j<k *)
                           main := main + 1)
        endtest;
        while true do
E4:       test V until
            Sj setto S0: buff := buff + j;   (* 0<j<k *)
            AE setto S0: exit
          endtest
      end;
E5: test V until          (* choose next controller *)
      Sj setto ELECT: (buff := buff + j;      (* 0<j<k *)
                       main := main - 1)
    endtest;

  LEAVING:                               (* begin leaving protocol *)
    while main > 0 do
E6:   test V  until
        Sj    setto S^ : buff := buff + j;   (* 0<j<k *)
        QUERY setto ONE: main := main - 1
      endtest;
    while true do
E7:   test V until
        Sj    setto S0: buff := buff + j;   (* 0<j<k *)
        QUERY setto S0: exit
      endtest;
    while buff > 0 do
E8:   test V until
        S0    setto S1: buff := buff -1
      endtest;
E9: test V until S0 setto BYE endtest;
    goto REMAINDER
  end;
```

Program for Lockout-free Mutual Exclusion

Figure 4-2 (continued)

here.

Some liberties are taken with the above syntax in the program given in the next section. In particular, not every item in the <set-list> is given explictly when it is convenient to group several of them together. A comment is given in each case which indicates how the condensed code could be expanded to the formal syntax of Figure 4-1.

There are other non-standard features in the program but these should cause no problems for the reader. For example, the "exit" statement is used to escape from the closest enclosing "while" loop. Also note that parentheses are sometimes used instead of "begin" and "end".

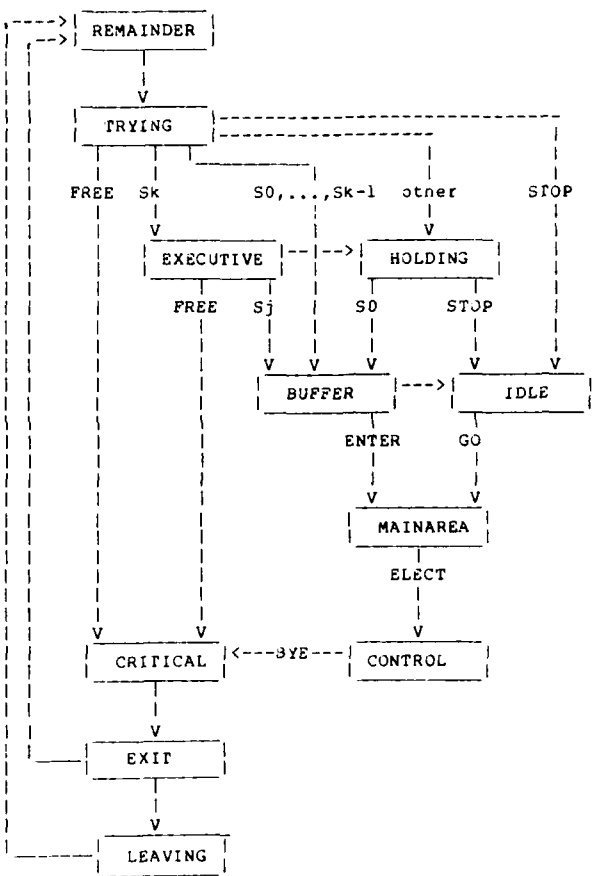
The reader should have no difficulty in interpreting program A in Figure 4-2 in terms of the formal model. For consistency, assume that all "local" steps of a process (*i.e., those that do not access the shared variable*) are included in the transition of the <u>preceding</u> test-and-set. Then the only values that the location counter of a program may take on are those labeled by "Tn" or "En". If the value of the program counter corresponding to program state Xi(q) is Tn or En, then program i is said to be <u>at location</u> Tn or En, respectively, at q.

<u>An Algorithm for Lockout-free Mutual Exclusion</u>

A critical (M,N)-system, $S = (V,X,p,q_0)$, which satis-

fies lockout-free mutual exclusion is defined as follows. The program for each process is given in Figure 4-2. Note that the definition of program A depends on N. Since only one shared variable is required, M=1, and V is used rather than V_1 in Figure 4-2. The initial id of S, q_0, is defined so that $V(q_0)$ = FREE and $Xi(q_0)$ = *the beginning of program* A for all i $\in$ [N].

The overall flow of the algorithm is given in Figure 4-3. The values shown on the lines of the figure indicate the value which V must have to allow the indicated transition. Upon leaving its remainder region, each process executes a protocol in its trying region to determine when it will enter its critical region. If the shared variable happens to have value "FREE", the critical region is empty, so the process may go immediately to its critical region. After leaving the critical region, the process executes a protocol in the exit region to help select the next process to go critical. During the execution of the protocols, one process has special importance and is designated the "controller". A process which is at location T8, T9, E0, E1, E2, E3, E4 or E5 is the current controller (the protocols guarantee that there is at most one controller at any id -- if V=FREE, then no controller exists). The controller has the responsibility to shepherd the other processes through the trying region. The last



High Level Flowchart of Program A

Figure 4-3

act of the current controller is to select the next controller. The former controller then becomes the "leaving" process. The leaving process must send certain information to the new controller before the leaving process may return to its remainder region. If there is no process waiting in the trying region, the old controller simply sets V to FREE, indicating that the critical region is free.

The shared variable may take on k+11 values: S0,S1,...,Sk,FREE,STOP,ENTER,AE,GO,AG,ELECT,QUERY,ONE and BYE, where k = ⌊N/2⌋. (Note: the program in Burns, et al. [BFJLP78] uses only k+9 values. Two additional values are used here to simplify the proof. The program in the referenced work can be reconstructed by literally replacing all occurences of constants AE, AG and BYE by constant ACK.) Values S0,S1,...,Sk are <u>counting values</u> and the others are <u>message values</u>. The counting values are used to keep a count of the number of processes which have entered their trying regions since the controller last examined the shared variable. But there are not enough counting values to keep an accurate count when more than k processes enter their trying regions one after the other. <u>Wraparound</u> occurs when the value of V is changed from Sk to S0 by a process entering its trying region at T0. The process which makes this transition is called the "executive".

After a wraparound, the controller cannot obtain an accurate count of the number of processes in the trying region. The executive is responsible for eventually correcting the discrepancy in the controller's count.

The controller and the executive communicate with other processes in the system by "sending messages". All message values except STOP are called <u>controller messages</u>. To send a message, V is set to a message value which is expected by a target process. The sending process must be sure that there is at least one (target) process which is waiting for the message. The target process responds to ELECT, GO or QUERY by changing V to AE, AG, ONE or BYE and to STOP by changing V to S0. The sending process is then able to detect that the message has been received and may send another message. If a process enters the trying region while V has as its value a controller message, the entering process cannot leave V unchanged (lest it be locked out). In this situation, the entering process "holds" the message in its local variable M and later puts the held value back into V. The executive may also have to hold a controller message temporarily. The system guarantees that a message will only be held for a finite number of steps before it is detected by an appropriate process.

A sketch of a formal proof of correctness is given in the next section. To give the reader a feel for the opera-

tion of the program, a description of one possible execution is given below. The execution is broken into three parts. In Part 1, wraparound does not occur, and no messages are held. In Part 2, wraparound is allowed, but there is still no holding of messages. In Part 3, holding is examined.

For Part 1 assume that wraparound does not occur, so no process ever enters the sections of code labeled EXECUTIVE or IDLE and no process sets its local variable "idlers" to a value other than zero. ("Idlers" keeps track of the number of processes which have been sent to the location labeled IDLE.) Also, assume that no process at location T0 happens to take a step when V is not a counting value, so holding does not occur.

From the initial id, the first process which enters becomes the controller. While this process is controller, additional processes enter their trying regions and go to the BUFFER. The controller, finding that MAINAREA is empty and BUFFER is not, moves all the processes in BUFFER to MAINAREA. Then each process in MAINAREA in turn is allowed to execute a critical region until the MAINAREA is emptied. No lockout is achieved because every process will eventually reach MAINAREA when BUFFER is emptied, and every process in MAINAREA will eventually reach its critical region. The following paragraphs discuss Part 1 of the

computation in more detail.

The process which takes the first step from the initial id will find V=FREE and immediately execute its critical region, moving to location E2 and setting V=S0. If no other process takes a step in the meantime, the process at E2 (the controller) will find V still equal to S0 on its next step. In this case, it will return immediately to its remainder region and set V=FREE, returning the system to its initial id.

Assume that j_1 processes ($0<j_1 \leq k$) execute one step each after the controller reaches E2. Each of these process moves to location T5, the BUFFER. Then, when the controller takes is next step, it will find $V=Sj_1$. It will then set its local variable $buff=j_1$ and go to E3. The controller then executes the loop containing E3 and E4 at least j_1 times. In each iteration, an ENTER message is sent to a process in BUFFER, the local variables buff and main of the controller are decremented by one and incremented by one, respectively. The target process responds by setting V to AE and going to T7, MAINAREA. Suppose at some point right after the execution of E4 (so that V=S0) and while buff is still greater than zero, j_2 ($0<j_2 \leq k$) additional processes which are still at T0 take one step each. Then the controller's next iteration of E3 will increment buff by j_2, causing the loop to be executed

at least j_1+j_2 times. Since at most N-1 process can move from T0 during the execution of the loop, eventually the controller will reduce its buff variable to zero, increase main to j_1+j_2 and go to location E5. Note that an id at which the BUFFER is empty occurs in the computation. Therefore, at this point, all of the processes in BUFFER have moved to MAINAREA and MAINAREA is not empty.

Now the controller sends a single ELECT message, decreases main by one, and moves to E6 (main is still greater than zero since $j_1+j_2 > 1$). Any process at location E6, E7, E8 or E9 is not a controller, but is called the leaving process. The next process in MAINAREA to take a step will see V=ELECT and move to T8, becoming a new controller. The new controller then executes the loop containing T8 and T9 j_1+j_2 times. For the first j_1+j_2-1 iterations, the leaving process responds to the QUERY messages by setting V=ONE. This effectively transfers the count of the processes in the MAINAREA to the new controller. On the final iteration, the leaving process responds to the QUERY message by setting V to BYE and going to its remainder region. The new controller then detects V=BYE, executes its critical region and goes to E5.

Each controller in turn receives the count of the processes still in the MAINAREA from the leaving process, executes its critical region, selects a new controller from

the MAINAREA and becomes a leaving process which sends the count of MAINAREA to the next controller.

At various points during the execution, the controller (and the leaving process while it is at E6) may detect a value of $V=Sj_3$ $(0<j_3\leq k)$, indicating that j_3 processes have moved to BUFFER. This count is passed on by the leaving process in the following way. On the final iteration of the T8-T9 loop, the leaving process does not respond to the QUERY message with BYE immediately, but sets V=S0 and the executes the E7 loop j_3 times. Each iteration of the loop reduces the local buff variable of the leaving process by one and changes V from S0 to S1. The controlling process, meanwhile, looping at location T9, changes S1 back to S0 and increments its local buff variable by one. Note that if j_4 additional processes enter from T0 at some point, the amount carried in the buff variable will be increased to j_3+j_4.

Now consider what happens when the last process in MAINAREA completes the T8-T9 loop. Its main variable will have value 0, so it will go to location E3 rather than E5 after executing its critical section. The value of the local buff variable is j_3+j_4. This situation is essentially the same as when the first process to enter the system reached E3. This ends Part 1 of the execution.

Part 2 begins where Part 1 left off. The controller is executing the loop at E3-E4, moving the processes in BUFFER to MAINAREA. At some point right after the controller executes E4 and while the controllers buff variable is still greater than zero, k+1 processes take one step each from T0. This returns V to the value S0, so the controller cannot detect that anything has happened. The k+1st process to enter, however, saw V=Sk and knows that k "extra" processes are in the BUFFER. This process moves to T1 and becomes the executive.

Before the executive takes another step, the system may fill and empty MAINAREA many times. However, when the BUFFER is moved to MAINAREA, k processes will be left in the BUFFER. When the executive takes its first step, it sends a STOP message. This message will be received either by a process in the BUFFER or by a process entering its trying region from T0. Since an entering process sets V from STOP to S1, the effect of the STOP message is to move one of the "extra" process to the IDLE location (T6) and to reduce the discrepancy between the buff count of the controller and the actual number of processes in the BUFFER.

The executive will send at least k STOP messages. While doing this, it may see $V=Sj_5$ $(0<j_5\leq k)$ rather than S0. The executive then "picks up" the addition j_5 processes

which are unknown to the controller and will send a total of $k+j_5$ STOP messages. When the executive finishes executing the loop at T1, the controller's count of the number of process in the BUFFER is correct. The executive then executes statement T3 and moves to the BUFFER like an ordinary process. (Note that the executive may find V=FREE. In this case, there is no controller to move the executive to critical, so the executive simply goes directly to critical and becomes the controller itself.) Since at least k processes will remain at IDLE until the executive reaches its critical region, another wraparound cannot occur while the executive is in its trying region. Therefore, the executive will eventually be moved to the MAINAREA and then become the controller.

After the executive becomes the controller and finishes the T8-T9 loop, it will execute its critical region and move to E0. The E0-E1 loop is used to move all $k+j_5$ process from IDLE to MAINAREA. The controller then moves to E3 or E5, depending on the value of its buff variable. The system will then allow each process in MAINAREA to reach its critical region in turn. Since no new executive can reach MAINAREA until MAINAREA is emptied, the processes in MAINAREA cannot be locked out. This ends Part 2.

So far, processes have only entered from T0 when V happened to have a counting value. Part 3 examines the behavior of the system when a process enters from T0 and V has a value of ENTER,AE,GO,AG,ELECT,QUERY,ONE or BYE. V only takes on these values during the sending of a message. Until the acknowledgement of the message is received by the sending process, the sending process will be cycling at one of the "grounding" locations, T9, E1, E4, E6 or E7. At any of these locations, whenever the sending process finds that V has a value of Sj ($0<j\leq k$), V is set to S0 and the local buff variable of the sending process is incremented by j. This behavior is called grounding V. The sending process keeps grounding V until the acknowledgment is received. Suppose a process enters from T0 while V is equal to one of the indicated values. Then the entering process will go to location T4 and hold the message value until it detects that V=S0 or V=STOP. This must happen withing a finite number of steps because some process will continue to ground V while the message is being held. After each grounding, V=S0, and V can be changed from S0 only by a process entering from T0 or by an executive. But the executive will only set V to STOP, S0 or (once only) to S1, so the executive will not keep the holding process from releasing its message. Since a process can only move from T0 once while a message is being held (holding the message

prevents processes from moving through their critical regions), the value of V will eventually remain constant at S0, and the holding process can reset V to the held value. A message can be held only a finite number of times before it is delivered because each holding requires a distinct process (which has just left T0).

The executive may also hold a message. However, the executive will only hold one message at T2, and this message will be released within a finite number of steps for the same reason that an entering process eventually releases its held message. Therefore, the holding of messages has no affect on the correctness of the system, and the system is lockout-free.

Correctness of the Algorithm

The correctness of a complex algorithm involving asynchronous processes is always suspect because of the many of cases that may occur during its execution. Unfortunately, a complete, formal proof of correctness of the algorithm based on program A would probably double the length of this thesis. Also, it is not clear that a very long, low level proof would be convincing [DLP79]. Therefore, only a suggestive series of lemmas with proof sketches will be given.

The first three lemmas prove properties about the

algorithm which are true at all ids which are reachable from the initial id. The first lemma lists a number of facts which will be useful in later proofs and also shows that the algorithm satisfies mutual exclusion. The next lemma gives facts about the state of the local variables of a process at certain locations in the program. The third lemma includes the key fact that two executives cannot exist simultaneously. All three lemmas may be proved by straightforward induction.

Lemma 4.4 shows that the algorithm is deadlock-free. The proof requires reasoning about the behavior of the algorithm during infinite computations. The final lemma shows that the algorithm is lockout-free, which is sufficient together with Lemma 4.1 to show that the algorithm is correct.

In the following, $S = (V,X,p,q_0)$ is the critical (1,N)-system such that V = {S0,S1,...,Sk,FREE,STOP,ENTER, AE,GO,AG,ELECT,QUERY,ONE,BYE}, each X_i for $i \in [N]$ corresponds to the states of program A, each p.i for $i \in [N]$ is defined so that its transitions correspond to program A, and q_0 is defined so that $V1(q_0)$ = FREE and Pi is at location T0 at q_0 for each $i \in [N]$.

For any location L of S and any id q of S let L(q) = { $i \in [N]$: Pi is at location L at q}. Note that the convention that local steps are combined with the preceding

test-and-set implies that T0(q) = the set of processes in their remainder region at q. (This is only because the apparent loop at T0 is never executed.) For any $i \in [N]$, let $CE_i = C_i \cup E_i$. If $Xi(q) \in CE_i$ then process i is in its CE region at q. Let CE(q) = E0(q) U E1(q) U ... U E9(q). The set of processes in CE(q) are exactly those which are in their CE regions at q. Also, let CN(q) = T8(q) U T9(q) U E0(q) U E1(q) U ... U E5(q). CN(q) is the set of processes which are controllers at q. If $i \in$ CN(q) the Pi is in its CN region at q.

For any id q of S and $i \in [N]$, let $M_i(q)$ be the value of local variable M of process i at id q. If $M_i(q) \neq S0$, process i is holding message $M_i(q)$ at q. Let TV(s)=1 if s is a true statement and TV(s)=0 otherwise. For any message value, Y, let Y(q) = TV(V(q)=Y) + TV(M_1(q)=Y) + TV(M_2(q)=Y) + ... + TV(M_N(q)=Y). (For example, ENTER(q) = 0 if and only if V(q)≠ENTER and no process is holding an ENTER message.) Each Y is a message function. The value of Y(q) represents the number of Y messages present in the system at id q. (Actually, as shown by assertion g of Lemma 4.1, for any reachable id q, Y(q)≤1.) Let CMSG(q) = FREE(q) + ENTER(q) + AE(q) + GO(q) + AG(q) + ELECT(q) + QUERY(q) + ONE(q) + BYE(q).

For any id q of S and any $i \in [N]$, let $buff_i(q)$, $main_i(q)$, and $idlers_i(q)$ be the value of Pi's local

variables buff, main and idlers, respectively, at q. Also let buff(q) = $buff_1(q) + buff_2(q) + \ldots + buff_N(q)$, main(q) = $main_1(q) + \ldots + main_N(q)$, and idlers(q) = $idlers_1(q) + \ldots + idlers_N(q)$. Let S(q)=j if V(q)=Sj, $0 \leq j \leq k$, and 0 otherwise.

Lemma 4.1

For any id q of S which is reachable from q_0, assertions a through k are true.

a. |CE(q)| + FREE(q) + BYE(q) = 1

b. |CN(q)| + FREE(q) + ELECT(q) = 1

c. ENTER(q) + AE(q) = |E4(q)|

d. GO(q) + AG(q) = |E1(q)|

e. QUERY(q)+ ONE(q)+ BYE(q)+ |E8(q)|+ |E9(q)| = |T9(q)|

f. ELECT(q) = |E6(q)| + |E7(q)|

g. CMSG(q) + |T8(q)| + |E0(q)| + |E2(q)| + |E3(q)|
+ |E5(q)| + |E8(q)| + |E9(q)| =1

h. buff(q) = |T4(q)|+ |T5(q)| -S(q) -ENTER(q) -STOP(q) ≥0

i. main(q) = |T7(q)|+ ENTER(q)+ GO(q) -ELECT(q) -ONE(q) ≥0

j. idlers(q) = |T6(q)| + STOP(q) - GO(q) ≥0

k. STOP(q) ≤ |T1(q)| + |T2(q)| + |T3(q)| + |E0(q)|

Proof sketch: It is easy to see that all the assertions are true at q_0. Let q and q' be ids of S and $i \in [N]$ be such that $q \xrightarrow{i} q'$. It is only necessary to show for each assertion that if the assertion holds at q then it also holds at q'. The lemma then follows by induction on

the length of schedules.

None of the arguments is particularly difficult. Assertion a is proved here and the remainder are left to the reader.

Since |CE(q)|, FREE(q) and BYE(q) are non-negative integers, one of the following three cases must hold.

Case 1: |CE(q)|=1, FREE(q)=0 and BYE(q)=0. Pi cannot enter the CE region in the transition q ->q' since all such transitions require that V(q)=FREE or BYE. If Pi does not leave the CE region in q->q', then CE(q')=CE(q) and V(q') cannot be FREE or BYE since these values can only be set by a transition leaving the CE region. If Pi does leave the CE region in q->q', then it must be at E2 or E9 at q and set V(q')=FREE or or BYE, respectively. For every possibility, assertion a is true at q'.

Case 2: |CE(q)|=0, FREE(q)=1 and BYE(q)=0. Pi cannot be in the CE region at q, so the transition q->q' cannot make FREE(q') or BYE(q') greater than FREE(q) or BYE(q), respectively. If Pi does not go to the CE region in q->q', then |CE(q')|=0 and FREE(q')=1. If Pi does go to the CE region in q->q', then Pi must be at T0, T1 or T3 at q, V(q)=FREE and V(q')=S0, so that |CE(q')|=1 and FREE(q')=0. Again, assertion a is true at q' for every possibility.

Case 3: |CE(q)|=0, FREE(q)=0 and BYE(q)=1. Suppose V(q)≠BYE. FREE(q)=0 implies that V(q)≠FREE. Therefore, Pi cannot enter the CE region from q so CE(q')=CE(q). The only way that BYE(q') could differ from BYE(q) is for the transition from q to q' to set V to BYE. The only transitions which could do this require that M_i(q)=BYE and i ∈ T2(q) U T4(q). But then BYE(q')=BYE(q), so assertion a is true at q'.

On the other hand, suppose V(q)=BYE. Then the transition to q' cannot change the truth of assertion 4.1 unless Pi is at location T9 at q. But if Pi is at location T9 at q, then Pi is critical at q' and V(q')=S0, so |CE(q')|=1, FREE(q')=0, BYE(q')=0, and assertion a is true at q'. □

The next lemma lists some facts which are useful in reasoning about lockout.

Lemma 4.2

Let q be an id of S which is reachable from q_0, and let i ∈ [N]. Then if Pi is at location T1 at q, $main_i(q)$ = 0. If Pi is at location T0, T2, T3, T4, T5, T6 or T7 at q then $main_i(q)$ = 0 and $buff_i(q)$ = 0. Finally, if Pi is at location T0, E2, E3, E4, E5, E6, E7, E8 or E9 at q, then $idlers_i(q)$ = 0.

Proof: These facts hold by the flow of the program and the exit conditions of the loops at E6, E8, T1 and E0-E1. □

For any id q of S, let Exec(q) = { i ∈ [N] : $idlers_i(q)$ > 0 } U T1(q) U E1(q). If i ∈ Exec(q), then Pi is said to be an executive at q. The next lemma shows that there may be at most one executive at any id and that no executive may reach location T6.

Lemma 4.3

Let q be an id of S which is reachable from q_0, and let i ∈ [N]. If i ∈ Exec(q), then assertions a, b and c hold.

a. $buff_i(q) + idlers_i(q) + main_i(q) \geq k$
b. Exec(q) = {i}
c. i ∉ T6(q)

Proof: Evidently all the assertions hold at q_0, since Exec(q_0) is empty. It is sufficient to show that for all ids q and q' of S such that $q \xrightarrow{i} q'$, if assertion a, b or c holds at q then assertion a, b or c, respectively, holds at q'.

Assertion a is shown first. Note that the only transition which increases the value of local variable idlers is from location T1. Then by Lemma 4.2, Pi must be at location T1, T2, T3, T4, T5, T6, T7, T8, T9, E0 or E1 at q. But no transition from any of these locations can decrease the value of buff+main+idlers. Thus, the only transition which could show that assertion a is false would

have i ∉ Exec(q) and i ∈ Exec(q'). But this implies that i ∈ T0(q) and i ∈ T1(q'), and this transition sets $idlers_i(q') = k$. Therefore, assertion a holds at all ids reachable from q_0.

For assertion b to be false, there must be a transition such that there is an integer j ∈ [N] for which j ∈ Exec(q), i ∈ Exec(q) and i,j ∈ Exec(q'). But then, by assertion a, $buff_i(q') + main_i(q') + idlers_i(q') + buff_j(q') + main_j(q') + idlers_j(q') \geq 2k \geq N$. Then by assertions h, i and j of Lemma 4.1, $|T4(q')| + |T5(q')| + |T6(q')| + |T7(q')| \geq N$. But, since i ∈ T1(q), this contradicts the fact that there are only N processes in the system, so assertion b holds at all ids reachable from q_0.

Suppose assertion c is false. Then there must be a transition such that i ∈ Exec(q), i ∉ T6(q) and i ∈ T6(q'). This can only occur if V(q) = STOP and Pi is at location T0, T4 or T5 at q. (Pi cannot actually be at T0 at q since it is an executive.) By assertion k of Lemma 4.1, there must be a j ∈ [N] such that Pj is at location T1, T2, T3 or E0 at q'. But then Pj is also an executive at q and j ≠ i, contradicting assertion b. Therefore, assertion c holds at all ids reachable from q_0. □

Let i ∈ [N] and q, q' be ids of S such that $q \xrightarrow{i} q'$. Pi <u>moves forward</u> in transition q ->q' if and only if there are locations L1 and L2 such that Pi is at location L1 at

q, Pi is at location L2 at q', L1 ≠ L2, and no process can move from L2 to L1 without reaching location T0. It is easy to see that Pi moves forward whenever it changes locations unless it moves from T8 to T9, E0 to E1, or E3 to E4.

<u>Lemma 4.4</u>

System S is deadlock-free.

<u>Proof sketch</u>: Suppose that S can be deadlocked, and let h be an infinite schedule of S which is R-admissible from q_0 and exhibits deadlock. Let $z_0 = q_0q_1\ldots$ be the id sequence from q_0 by h. By definition, there must be a point in z_0 after which no process changes regions. Since any process which continues to move forward must eventually change regions, there must also be a point after which no process moves forward. Choose an integer a≥0 such that no process moves forward in computation $z = q_aq_{a+1}\ldots$

Case 1: the value of V changes infinitely often in z. Since V is changing and no process is moving forward, there must be a process looping infinitely often at location T1, T8-T9, E0-E1, E3-E4, E6, E7 or E8. (Note: looping at T8-T9 includes looping at T9 and alternating between locations T8 and T9. E0-E1 and E3-E4 have similar meanings.)

A process looping at T1 will set V to STOP (the "other" branch can be taken only once by assertion g of

Lemma 4.1). But V can only be changed from STOP by a process which is moving forward, which is forbidden by the choice of a. Therefore, a process at T1 cannot assist in changing V infinitely often. Similar arguments show that a process looping at E0-E1 or E3-E4 cannot change V infinitely often in z.

The only transition which changes V from S0 to another counting value without causing a forward move is at E8. But, by assertion a of Lemma 4.1, no process can be at E8 while a process is looping at E6 or E7. Therefore, a process looping at E6 or E7 can contribute only a finite number of changes of V in z. This is also true for a process at E8.

The only remaining possibility is that a process is looping at T8-T9 in z. But an infinite number of iterations of the T8-T9 loop requires an infinite number of iterations of either the E6 or E8 loop, which has already been shown to be impossible in z. Therefore, Case 1 leads to contradiction.

Case 2: V changes finitely often in z. Then choose $b \geq a$ so that V is constant in $z' = q_b q_{b+1} \cdots$. Suppose V has a counting value in z'. By assertion g of Lemma 4.1, there must be a process at T8, E0, E2, E3, E5, E8 or E9 in z'. If a process is at E8 or E9, assertions a, b and g of Lemma 4.1 imply that there must also be a process at T8 or

T9. If Pi is at T9 and V≠S0, then the next step of Pi will change V, while if V=S0 then the next step of the process at E8 or E9 will change V, contrary to assumption. But if Pi is at T8, E0, E2, E3 or E5 in z', then the next step of Pi will change V, contrary to assumption.

Suppose V=STOP for every id in z'. Since buff(q) must be non-negative at every id q reachable from q_0 in S, assertion h of Lemma 4.1 implies that some process, Pi, is at T4 or T5 in z'. But Pi will move forward on its next step, contrary to assumption, so V cannot be equal to STOP. Similar arguments can be made to show that V cannot be any message value other than FREE.

Suppose V=FREE for every id in z'. By assertions a and b of Lemma 4.1, no process can be in the CN or CE region in z'. Also, by assertion g of Lemma 4.1, no process can be at location T2 or T4 in z'. If a process were at T1 or T3 in z' it would move forward on its next step, so all processes must be at locations T0, T5, T6 or T7 in z'. Then, by Lemma 4.2, buff(q) and main(q) must be zero for every id q in z', so, by assertions h and i of Lemma 4.1, no process can be at location T5 or T7 in z'. By Lemma 4.3, no executive can be at T6, so idlers(q)=0 for every id q in z', which implies that no process is at T6 by assertion j of Lemma 4.1. The only possibility is that all processes are at T0 at q_b. But then the next process to

take a step will move forward. This contradiction completes the proof. □

Lemma 4.5

System S is lockout-free.

Proof sketch: Let h be a (necessarily infinite) schedule which exhibits lockout from q_0, and let $q_0q_1\cdots$ be the id sequence from q_0 by h. Let $i \in [N]$ be such that Pi is locked out from q_0 by h. Choose an integer $a>0$ so that Pi does not move forward in computation $z = q_aq_{a+1}\cdots$ By definition, Pi cannot be at location T0 at q_a. Also, Pi cannot be in the CN or CE region at q_a, since this would deadlock the system and contradict Lemma 4.4. Pi cannot be at location T2 or T4 either since V changes value whenever a process changes regions. Therefore, Pi must be at location T1, T3, T5, T6 or T7 at q_a.

Suppose Pi is at location T1 at q_a. One way that Pi could be locked out would be for V to have value STOP every time Pi takes a step in z. Since only an executive can set V to STOP, and Pi is the only executive in z by Lemma 4.3, V must be equal to STOP for all of z. But this would deadlock the system, so Pi must see V≠STOP an infinite number of times in z. This implies that Pi's local variable idlers will increase without bound, which is impossible by assertion h of Lemma 4.1. Therefore, Pi cannot be at location T1 a q_a. Also, Pi cannot be at

location T3 at q_a since this would require that V be equal to STOP for all of z, which has already been shown to lead to contradiction. The remaining possible locations are T5, T6 and T7.

T7. Suppose that Pi is at T7 for all of z. Then $|T7(q_j)|>0$ for all $j\geq a$. By Lemma 4.4, an infinite stream of processes pass through the CE region in z. Suppose one of these processes reaches E2 or E3 at some id q_b, $b\geq a$. Then, by Lemma 4.2 and assertions a, b and g of Lemma 4.1, $|T7(q_b)|=0$, which is a contradiction. Therefore, V≠ENTER and V≠FREE at any id in z, which implies that no process can go to T7 from T5 and that no executive can go to the CE region from T1 in z.

Since any executive which enters from T0 in z can go no further than T5, there must be an integer $c>a$ such that no executive is in the CE region in $z' = q_cq_{c+1}\cdots$ Then V≠GO in z', so no process can enter T7 in z'.

Since processes passing through the CE region in z' will not execute statement E2, each process will execute E5 and send an ELECT message. Each ELECT message will cause a process to leave T7. But, since the number of processes at T7 is finite and non-increasing in z', every process at T7 (including Pi) must leave T7 in z', contradicting the assumption.

T5. Suppose that Pi is at T5 for all ids in z so that $|T5(q_j)|>0$ for all $j \geq a$. By the argument in the previous paragraph, there is an integer $b \geq a$ such that $main(q_b)=0$. Then there must be an integer $c \geq b$ such that for some $j \in [N]$, Pj is at E3 at q_c. Let d be the least integer greater than c such that Pj is at location E5 at q_d (d exists by Lemma 4.4). If no process is at T1 at q_d, $buff(q_d)=0$ by Lemma 4.2, and $|T5(q_d)|=0$ by assertions g and h of Lemma 4.1, which is a contradiction. Therefore there must be an integer $x \in [N]$ such that Px is at T1 at q_d.

Let e be the least integer greater than d such that Px is not at T1 at q_e (Px cannot be locked out at T1 by the earlier argument). If Px is in the CE region at q_e, then $buff(q_e)=0$, which again implies that $|T5(q_e)|=0$, a contradiction. Thus, there must be a least integer $f>e$ such that Px is at location T5 at q_f.

Now let g be the least integer greater than f such that for some $z \in [N]$, Pz moves from E4 to E5 in the transition $q_{g-1} \xrightarrow{z} q_g$. Since Px must be at T5 or T7 at q_g, assertion b of Lemma 4.3 implies that no process is at T1 at q_g. But then $|T5(q_g)|=0$, another contradiction. All possibilities lead to contradiction so, the supposition that Pi stays at T5 in z must be false.

T6. Suppose that Pi is at T6 for all ids in z. Let $b \geq a$ be the least integer such that some process Pj, $j \in [N]$, is at E2 or E5 at q_b. (Integer b exists by Lemma 4.4.) If no executive exists at q_b, then $idlers(q_b)=0$ which implies that $|T6(q_b)|=0$, a contradiction. Therefore, for some $x \in [N]$, Px is an executive at q_b. Since Px will not go to T6 by assertion c of Lemma 4.3, the previous paragraphs imply that it must eventually reach the CE region. Then there must be a least integer $c>b$ such that Px is at E1 at q_c and Px is not at E1 at q_{c+1}. But then $idlers(q_c)=0$ which implies that $|T6(q_c)|=0$ and that Pi cannot be at T6 at q_c. □

Theorem 4.3

System S (based on program A) is lockout-free and satisfies mutual exclusion.

Proof: Lemma 4.5 shows that S is lockout-free. Assertion a of Lemma 4.1 implies that S satisfies mutual exclusion since the critical region is a subset of the CE region. □

Theorem 4.4

There is a critical (1,N) system $S'=(V',X',p',q'_0)$, $N \geq 1$, that satisfies lockout-free mutual exclusion and is memoryless such that $|V| = \lfloor N/2 \rfloor + 9$.

Proof: Let S' be identical to S except that all occurences of constants AE, AG and BYE are literally replaced by a new constant, ACK. (This converts Program A into the form given in Burns, et al. [BFJLP78], as

mentioned in an earlier parenthetical comment.) Clearly S' is a critical (1,N)-system which is memoryless and |V'| = ⌊N/2⌋ + 9. Let f map every state in S into the corresponding state of S' by replacing V with ACK if it is AE, AG or BYE and changing local variable M value from AE, AG or BYE to ACK. Then any schedule h of S is also a schedule of S'.

Claim: $f(r(q_0,h)) = r'(f(q_0),h)$, where r' is the result function of S'. If not then there is an integer i ∈ [N] and a reachable id q of S such that $f(r(q,i)) \neq r'(f(q),i)$. Clearly, the only possibility is that V(q) = AE, AG or BYE and the state of Pi at r(q,i) does not correspond to r'(f(q),i). The only transitions for which this might be true are in the CE region. For example, Pi might be at location E1 at q and f(q). If V(q)=AG, Pi would not change locations in S but would in S'. But assertion d of Lemma 4.1 shows that this possibilities (and the other similar ones) cannot occur at ids of S which are reachable from q_0. Therefore the claim holds.

It should be clear that every computation of S' has a corresponding computation in S. By the claim, the region changes in both systems are matched exactly, so S' satisfies lockout-free mutual exclusion. □

If k in Program A is changed from ⌊N/2⌋ to N-1, then a wraparound transition cannot occur (by assertion a of

Lemma 4.3 and the fact that whenever all processes are in their remainder regions, V=FREE). Then no process can reach T1, T2, T3 or T6 and V cannot take on values STOP or GO. Let Program B be the program formed by setting k to N, removing the statments which include T1, T2, T3 and T6, and deleting the line in statement T0 which references Sk and the lines in T0, T4 and T5 which reference STOP. Also remove any references to STOP and GO in the type definitions and comments. Finally, change all occurences of AE, AG and BYE to ACK and equate Sk with FREE.

Theorem 4.5

There is a critical (1,N)-system, $S''=(V'',X'',p'',q''_0)$, which satisfies mutual exclusion and 1-bounded waiting such that |V''| = N+5.

Proof: Let S'' be the system corresponding to Program B, described above. S'' is a critical (1,N)-system, and V takes on at most N+5 values. All the lemmas proved for system S apply to system S'', but no wraparounds may occur. Therefore, whenever V=FREE, all processes must be in their remainder regions (at T0). This is the reason that FREE may be equated with Sk.

Suppose that S'' does not satisfy 1-bounded waiting. Then there must be a schedule h of S, i ∈ [N] and an id q of S reachable from q_0 such that process i 2-waits in comp(q,h). Let j ∈ [N] be such that Pj cycles two times in

comp(q,h), and let integer m and n be such that $1 \leq m < n$, $q_m \xrightarrow{1} q_{m+1}$, $q_n \xrightarrow{j} q_{n+1}$ and Pj is at T0 at q_m and q_n. Note that Pj must be at T4 or T5 at q_{m+1} since $V(q_m) \neq FREE$.

Now Pi cannot be in the CN region or the CE region at q_m, since it would have to change regions before q_n is reached in comp(q,h). Pi cannot be at T7 at q_m because every process at T7 must change regions before Pj can move from T4 or T5. Therefore Pi must be at T4 or T5 at q_m.

Since Pj moves to T7 in the computation $z = q_m q_{m+1} \ldots q_n$ there must be a point in z at which some process is in the E3-E4 loop. But when this loop terminates at some id q_a, $m < a \leq n$, $buff(q_a) = |T4(q_a)| + |T5(q_a)| = 0$, so Pi must have moved to T7 at q_a.

Now Pj is at T4 or T5 at q_{m+1} and Pi is at T7 at q_{m+1}. But there must be an id q_b, $b > m$, such that $main(q_b) = |T7(q_b)| = 0$ which occurs before Pj moves from T5. But this implies that Pi will change regions before Pj cycles two times, contradicting the assumption and proving that S" satisfies 1-bounded waiting.

Assertion a of Lemma 4.1 implies that S" satisfies mutual exclusion, so the theorem is proved. □

CHAPTER V

SYNCHRONIZATION OF MULTIPLE RESOURCES

The results in the preceding chapter addressed problems in which the asynchronous processes had to be excluded from simultaneously accessing a certain portion of their code referred to as the critical region. This may also be referred to as the "1-resource problem". The 1-resource problem is motivated by situations in which a resource cannot be safely accessed by multiple processes at the same time. For example, two processes updating the same database concurrently could introduce logical inconsistencies, although no inconsistency would arise if the processes execute their updates in some sequential order.

Another type of exclusion is required when processes share a pool of equivalent input/output devices, such as tape drives. Such devices must normally be dedicated for the use of one process at a time; however, any resource in the pool may be used to satisfy a request. If there are n resources in the pool, this is called the "n-resource problem", and the property required is called "n-exclusion". In this section, the n-resource problem is discussed informally in order to motivate the formal

definitions of the next section.

A system which satisfies n-exclusion will allow n processes (but no more) to be critical at the same time. To avoid degenerate solutions, up to n-1 processes must be allowed to stop in their critical regions without blocking any other process. This property is called "avoiding n-deadlock".

One way to construct an algorithm for the n-resource problem is to reduce the problem to a 1-resource problem and apply known solutions (such as those given in Chapter IV) to the latter problem, using an n-valued semaphore for exclusion. A solution of this kind is called the "bank" solution in the following, in analogy to the technique commonly used in banks. (Note: I originally developed a transformation for eliminating the exit region from mutual exclusion algorithms. Mike Fischer [FLBB79] generalized the technique to the bank algorithm described here.) In a bank, a single queue of customers waiting for service by several tellers is often used. The person at the head of the queue checks to see if one or more tellers are free, and if so, goes to any free teller.

The bank solution may be implemented for asynchronous processes by using a solution to the 1-resource problem (such as Programs A and B in Chapter IV) to select processes one at a time. The selected process waits (if necessary) until at least one resource is free, and then goes to its critical region. The count of the number of processes which hold resources (i.e., processes which are in their critical regions) is kept in an additional variable which ranges from zero to n. Note that, since test-and-sets are used for accessing the shared variables, the added variable may be combined with the existing shared variable, if desired. More precisely, the transformation of the programs in Chapter IV is described in Figure 5-1. W is the new shared variable which is initialized to zero and ranges from zero to n. This transformation is used to prove Theorem 5.1, which shows that an n-resource problem can generally be reduced to a 1-resource problem with an increase in the shared memory size of only a factor of (n+1). A corresponding lower bound is proved in Theorem 5.3.

The bank solution has a rather subtle defect which becomes apparent when several tellers become free at the same time. If $m \geq 2$ tellers are free, it would be desirable for the next m people in the queue to go immediately to m tellers. The bank solution requires them to file past the head of the queue one at a time. If the person at the head of the queue is very slow, others are slowed down unnecessarily. Indeed, if the person at the head of the queue "fails", then the whole system becomes blocked.

1. Replace all "goto CRITICAL" statements with "goto SELECT".

2. Replace the statement labeled "CRITICAL" by:

```
SELECT:
  test W until
    j setto j+1        (* 0≤j<n *)
  endtest;
```

3. Replace all "goto REMAINDER" statements with "goto CRITICAL".

4. Insert the following at the end of the algorithm.

```
CRITICAL:              (* critical region *)
  test W until
    j setto j-1        (* 0<j≤n *)
  endtest;
```

Bank Transformation (Fischer)

Figure 5-1

Note that the notion of failure used here does not imply any detectable malfunction. A failed process simply stops taking steps. This is quite different from the kind of failure considered by Rivest and Pratt [RP76] and Peterson and Fischer [PF77]. In these papers, when a process fails it goes to a predetermined state and sets a shared variable to a value indicating failure. For the type of failure used here, it is impossible to determine, in any finite portion of a computation, whether a process has failed or is only running very slowly.

An algorithm is "m-robust" if it continues to operate properly (processes trying to change regions eventually do so with the appropriate fairness conditions) while fewer

than m processes fail in their trying or exit regions. Theorem 5.2 states that there is an algorithm which is m-robust and uses only O(N) values of shared memory for synchronizing N processes. A corresponding lower bound is given in Theorem 5.4.

The fairness conditions defined in Chapter II are not compatible with the concept of robustness. A failed process will never change regions, so any fairness condition which depends on this (such as bounded waiting) cannot be maintained if other processes are not to be blocked forever. New fairness conditions are therefore defined in terms of processes becoming "enabled" to change regions, rather than in terms of actual region changes. A process is "enabled" when it can change regions without waiting for any other process, and cannot be blocked by any other process. The key distinction between enabling and actual region changing, which is exploited by the algorithm upon which Theorem 5.2 is based, is that a process can become enabled without taking any action of its own. Thus, failed processes can be made to "make progress" through the actions of the other processes in the system.

The lower bounds results (Theorems 5.3 and 5.4) given here were developed independently by myself, although they have already appeared as part of joint work with M. J. Fischer, N. A. Lynch and A. Borodin [FLBB79]. I

wish to thank my co-authors for help in polishing the results. The definitions given here are similar but not identical to those in the cited work.

Formal Definitions

In the following definitions, m, n, M and N are *positive integers, b is a non-negative integer and S is a* critical (M,N)-system.

If h' and h" are schedules such that h = h'h" and i does not occur in h", then final(i,q,h) = r(q,h'). If Pi does not halt in h then final(i,q,h) is undefined. A schedule h of S is m-admissible from id q of S provided $|\{i \in [N] : Pi \text{ halts in } h \ \& \ final(i,q,h) \in T_i \cup E_i\}| < m$. Note that "1-admissible" is equivalent to "R-admissible" defined in Chapter II, except that processes are allowed to stop in the critical region as well as in the remainder region. Intuitively, a schedule is m-admissible if less than m processes fail in the trying and exit regions.

For any id q of S let $T(q) = \{ i : Xi(q) \in Ti \}$, $C(q) = \{ i : Ci(q) \in Ci \}$, $E(q) = \{ i : Ei(q) \in Ei \}$ and $R(q) = \{ i : Ri(q) \in Ri \}$. An id q of S violates n-exclusion *if* $|C(q)| > n$. *S satisfies n-exclusion if no* id of S reachable from q_0 violates n-exclusion. The critical region is said to be full (when S satisfies n-exclusion) at any id q of S such that $|C(q)| = n$. Note

that "1-exclusion" is equivalent to "mutual exclusion" defined in Chapter II.

The following series of definitions lead up to the definition of "(m,n)-deadlock-free", which is needed for Theorem 5.2 and Theorem 5.4. Theorem 5.2 is included only to give a counterpoint to the lower bound of Theorem 5.4. For motivation and explanation of the following definitions, the reader is urged to consult Fischer, et al. [FLBB79].

Let q be an id of S and G be a subset of T(q) (respectively, of E(q)). G is C-group-enabled (respectively, R-group-enabled) at q provided for all schedules h in which each i ∈ G appears at least once, at least |G| distinct processes go directly from trying region to critical region (respectively, from exit region to remainder region) in comp(q,h). (Thus, a process not in G may prevent one in G from making a region change by making a change in its place, but this is all the damage such a process can do.) C-allocation(q) (respectively, R-allocation(q)) = max { |G| : G is C-group-enabled (respectively, R-group-enabled) at q}.

Let q be an id of S and h be a schedule of S. Schedule h exhibits (m,n)-deadlock from q provided a through d hold. Let z = comp(q,h).

a. h is infinite and m-admissible *from* q.

b. No process changes regions in z.

c. C-allocation and R-allocation do not change in z.

d. At least one of d1 and d2 holds.

d1. |T(q)| > C-allocation(q) and

C-allocationq) + C(q) < n.

d2. |E(q)| > R-allocation(q).

S is (m,n)-deadlock-free provided there do not exists an id q of S reachable from q_0 and a schedule h of S such that h exhibits (m,n)-deadlock from q.

Let i ∈ [N]. Pi is critical-enabled (respectively, remainder-enabled) at an id q of S provided for all finite schedules h of S not containing i, Pi is in its critical region (respectively, remainder region) at r(r(q,h),i). Let CEn(q) (respectively, REn(q)) denote { i ∈ [N] : Pi is critical-enabled (respectively, remainder-enabled) at q}.

A finite schedule h of S is an enabling schedule for Pi from q if i ∉ CEn(q) U REn(q) and i ∈ CEn(q') U REn(q'), where q' = r(q,h). Pi becomes enabled in comp(q,h) provided $h = h_1h_2h_3$ with h_2 an enabling schedule for Pi from $r(q,h_1)$.

Pi b-waits for enabling for Pj in comp(q,h) provided i ∈ (T(q) - CEn(q)) U (E(q) - REn(q))), Pi does not change regions or become enabled in comp(q,h') for any prefix h' of h and Pj cycles b times in comp(q,h).

S satisfies b-bounded waiting for enabling provided there do not exist an id q of S reachable from q_0, a schedule h of S and i and j ∈ [N] such that Pi (b+1)-waits for enabling for Pj in comp(q,h).

A system S is order preserving provided that the order of entry to the critical region is the same as the order of return to the remainder region for all computations from q_0. S has null exit regions if E_i is empty for all i ∈ [N].

Upper Bounds

Two theorems are given in this section which provide upper bounds on the number of shared values required to solve the generalized exclusion problem for bounded waiting and bounded waiting for enabling. The next section given corresponding lower bounds.

Lemma 5.1

Let b, n, M and N be integers, $1 \le n \le N$, $1 \le M$ and $0 \le b$. Let $S = (V,X,p,q_0)$ be any critical (M,N)-system which is (1,1)-deadlock-free and order preserving. Let $S' = (VxW,X',p',q_0')$ be the system constructed by the transformation of Figure 5-1. Then S' has null exit regions, satisfies n-exclusion, and is (1,n)-deadlock-free. Furthermore, if S satisfies b-bounded waiting then so does S'.

Proof sketch: Clearly, S' has null exit regions.

Let W(q) be the value of shared variable W at id q of S'. It is easy to show that |C(q)| = W(q) for every id q reachable from q'_0, which implies n-exclusion since W is bounded above by n.

A total mapping, f, can be defined from the computations of S' from q'_0 to the computations of S from q_0 which preserves the order of process region changes. Suppose f(comp(q'_0,h')) = comp(q_0,h). Then h is the same as h' except that all steps which correspond to waiting for W to be less than n in S' and the single step that each process makes which reduces W as the process moves to its remainder region are removed. Since S is order preserving, every region change in comp(q'_0,h) has a corresponding region change in f(comp(q'_0,h)). This implies that S' is (1,n)-deadlock-free and also that if S satisfies 1-bounded waiting, then so does S'. □

Theorem 5.1 (Pischer)

Let n and N be positive integers, $1 \leq n \leq N$. There exists a critical (1,N)-system S'=(V',X',p',q'_0) which has null exit regions, satifies n-exclusion, is (1,n)-deadlock-free and satisfies 1-bounded-waiting such that |V'| = (n+1)(N+5).

Proof sketch: Let S=(V,X,p,q_0) be the system corresponding the Theorem 4.5 in Chapter IV. S satisfies 1-exclusion, is (1,1)-deadlock-free and satifies 1-bounded

waiting, and |V| = N+5. S also has the property of mutual exclusion over the union of the critical and exit regions, so S is order preserving. Apply Lemma 5.1 to S to find S' as required. □

The next theorem refers to an algorithm which continues to operate correctly as long as at most m-1 processes stop in the trying or exit region. This is accomplished by always having m processes maintain current copies of the shared system information. For more details, see Fischer, et al. [FLBB79].

Theorem 5.2 (Pischer, et al.)

Let m, n, M and N be positive integers, $1 \leq n \leq N$. There exists a critical (M,N)-system S=(V,X,p,q_0) such that S satisfies n-exclusion, is (m,n)-deadlock-free and satifies 1-bounded-waiting for enabling and such that |V| is O(N).

Lower Bounds

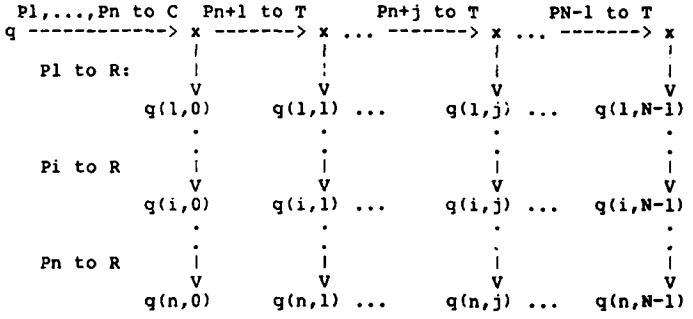
The next theorem give the lower bound corresponding to Theorem 5.1. The lower bound (n(N-n)) is quite close to the upper bound of the algorithm of the previous section ((n+1)(N+5)), so there is apparently little room for improvement with regard to shared space.

Theorem 5.3

Let b, n, M and N be integers such that $1 \leq n \leq N$, $1 \leq M$

and $0 \leq b$. Let $S=(V,X,p,q_0)$ be a critical (M,N)-system such that S has null exit regions, satisfies n-exclusion, is $(1,n)$-deadlock-free and satisfies b-bounded waiting. Then $|V| \geq n(N-n)$.

Proof: The theorem is trivial for $n=N$, so assume $n<N$. Let q be an id of S reachable from q_0 with all processes in their remainder regions (q exists since S is $(1,r)$-deadlock-free). Fix integers i and j, $1 \leq i \leq n$, $0 \leq j \leq N-n-1$. Construct a schedule as follows. From q, each of $P1,P2,\ldots,Pn$ in turn goes to its critical region (by the $(1,n)$-deadlock-free property) and stops. Next, each of $Pn+1,\ldots,Pn+j$ takes one step, moving to its trying region (by the n-exclusion property since the critical region is full). Then each of $P1,\ldots,Pi$ takes one step, going to its remainder region (since S has null exit regions). The resulting id is denoted by $q_{i\ j}$; it has $P1,\ldots,Pi$ and $Pn+j+1,\ldots,PN$ in their remainder regions, $Pi+1,\ldots,Pn$ in their critical regions and $Pn+1,\ldots,Pn+j$ in their trying regions. (See Figure 5-2.) In particular, the critical region is not full at $q_{i,j}$, and PN has not appeared in the described schedule from q to $q_{i,j}$.



Construction of Ids Used in Theorem 5.3

Figure 5-2

If all of the values $V(q_{i,j})$ are distinct, the theorem holds (i ranges over n values and j ranges over N-n values), so assume the contrary. There are two cases.

Case 1: $V(q_{i,j}) = V(q_{r,s})$ and $j < s$. Construct schedule h of S as follows. Starting from $q_{i,j}$, $Pn+1,\ldots,Pn+j$ go through critical regions to their remainder regions (by the $(1,n)$-deadlock-free property) and stop. Then PN cycles b+1 times from remainder to critical region. (again by the $(1,n)$-deadlock-free property).

But $q_{r,s}$ looks like $q_{i,j}$ to $Pn+1,\ldots,Pn+j$ and PN, so h causes the same behavior from $q_{r,s}$. Then Pj+1 (b+1)-waits for Pn and b-bounded waiting is violated.

Case 2: $V(q_{i,j}) = V(q_{r,j})$ and $i < r$. Construct a

schedule h as follows. Starting from $q_{r,j}$, i+1 of the processes in the set {P1,...,Pi} U {Pn+1,...,PN} move into their critical regions and stop. (There are sufficiently many processes because n<N.) This is possible because only n-r (< n-i) processes are critical at $q_{r,j}$ and no process other than those in the given sets is in its trying region at $q_{r,j}$. Since $q_{i,j}$ looks like $q_{r,j}$ to P1,...,Pi and Pn+1,...,PN, h causes the same behavior from $q_{i,j}$. But Pi+1,...,Pn are critical at $q_{i,j}$, so h applied from $q_{i,j}$ causes a violation of n-exclusion. □

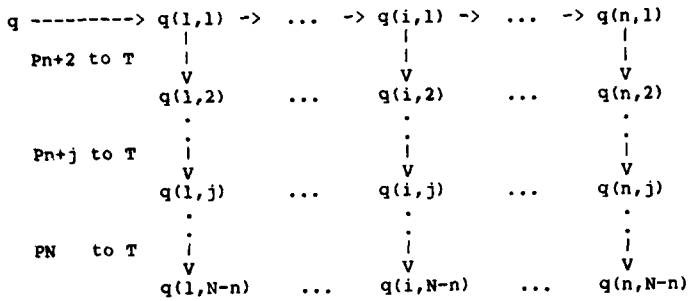
The next theorem gives the lower bound corresponding to Theorem 5.2. The two bounds are not nearly as close as in the previous case (the constant coefficient for the bound of Theorem 5.2 is exponential in each of m and n). Thus there is considerable room for improvement in the following result. Note that the value of m does not appear in any of the arguments.

Theorem 5.4

Let b, m, n, M and N be integers such that $1 \leq n \leq N$, $1 \leq m$, $1 \leq M$ and $0 \leq b$. Let $S=(V,X,p,q_0)$ be a critical (M,N)-system such that S satisfies n-exclusion, is (m,n)-deadlock-free and satisfies b-bounded waiting for enabling. Then $|V| \geq n(N-n)$.

Proof: The theorem is trivial for n=N, so assume n<N.

Let q be an id of S reachable from q_0 such that all processes are in their remainder regions at q. Define a "primary" schedule h and a sequence of ids $q_{i,1}$, $1 \leq i \leq n$, which appear in order in comp(q,h). Each $q_{i,1}$ has the i-1 processes P1,...,Pi-1 critical-enabled in their trying regions, Pi in its trying region, Pi+1,...,Pn+1 in their critical regions and all other processes in their remainder regions. Namely, starting at q, each of P2,...,Pn+1 in turn enters its critical region and stops. Then P1 takes one step, entering its trying region. The resulting id is $q_{1,1}$. Assume inductively that $q_{i,1}$ has been defined, i < n. Starting at $q_{i,1}$, both Pi+1 and Pi+2 leave their critical regions and go to their remainder regions without any other process taking a step (by the (m,n)-deadlock-free property). Then Pi+2 cycles b+1 times from remainder to critical, stopping in the critical region. This forces Pi to become critical-enabled (or b-bounded waiting for enabling would be violated). Then Pi+1 takes one step, entering its trying region (since n processes are already either critical or critical-enabled). The resulting id is $q_{i+1,1}$.



Construction of Ids Used in Theorem 5.4

Figure 5-3

Now fix integers i and j, $1 \le i \le n$, $1 \le j \le N-n$. Construct a "secondary" finite schedule as follows. Starting at $q_{i,1}$, each of Pn+2,...,Pn+j in turn takes one step, entering its trying region since n processes are either critical or critical-enabled at $q_{i,1}$. Call the resulting id $q_{i,j}$ (see Figure 5-3). Each $q_{i,j}$ has P1,...,Pi-1 critical-enabled in their trying regions, Pi+1,...,Pn+1 in their critical regions, Pi and Pn+2,...,Pn+j in their trying regions and all other processes in their remainder regions. If all of the $V(q_{i,j})$ are distinct the theorem holds, so assume the contrary. Their are two cases.

Case 1: $V(q_{i,j}) = V(q_{r,s})$ and $i < r$. P1,...,Pi are all critical-enabled in their trying regions at $q_{i,j}$, so that the schedule $h_1 = 12...i$ applied to $q_{i,j}$ moves

P1,...,Pi to their critical regions. None of P1,...,Pi takes a step either in the defined secondary schedule from $q_{i,1}$ to $q_{i,j}$, or in h from $q_{i,1}$ to $q_{r,1}$, or in the secondary schedule from $q_{r,1}$ to $q_{r,s}$. Thus, $q_{i,j}$ looks like $q_{r,s}$ to P1,...,Pi, and so h_1 has the same effect when applied from $q_{i,j}$. But Pi+1,...,Pn+1 are critical at $q_{i,1}$ and therefore also at $q_{i,j}$. Thus, h_1 applied from $q_{i,j}$ causes a violation of n-exclusion.

Case 2: $V(q_{i,j}) = V(q_{i,s})$ and $j < s$. Define a schedule h_1 as follows. Starting at $q_{i,j}$, all processes not in their remainder regions, P1,...,Pn+j, go to their remainder regions and stop. Then Pn cycles from remainder to critical b+1 times. Since $q_{i,s}$ looks like $q_{i,j}$ to P1,...,Pn+j, the behavior of these processes is the same when h_1 is applied from $q_{i,s}$. But Pn+s is in its trying region at $q_{i,s}$ and remains there throughout the application of h_1. Moreover, Pn+s is not critical-enabled at $r(q_{i,s},h_1)$ because the n processes P1,...,Pn are critical at this id. Thus b-bounded waiting for enabling is violated. □

Note that neither Theorem 5.3 nor 5.4 directly implies the other, although their statements are very similar. This is because b-bounded-waiting for enabling in Theorem 5.4 is more stringent than b-bounded-waiting in Theorem 5.3, whereas Theorem 5.3 includes the condition of null exit regions, which is not present in Theorem 5.4.

CHAPTER VI

SYNCRONIZATION IN A RING NETWORK

In single processor systems, the traditional measures for the performance of algorithms have been the amount of time and space required. In distributed systems, communications cost is also of interest. In this chapter, the number of messages required to solve a problem in a distributed system will represent communications cost. If all messages are about the same size, then this measure is comparable to costs that might be incurred on a packet switching communications network.

The network which connects together processes in a distributed system can be thought of as a graph (or multigraph). One important kind of network has a cycle as its related graph. This kind of network is called a ring and is the only type of network which will be considered here. The examination of network which correspond to more complex graphs will be left for future work.

Le Lann [LeL77] considers a system of asynchronous processes which communicate by passing messages. The processes are connected in a ring which allows messages to be sent in only one direction. (That is, each process can send messages to its left neighbor. A message which is

passed along from process to process will visit every process.) Mutual exclusion is provided in the system by a single control token which is circulated among the processes; only the process with the token can execute a critical section. Le Lann gives an algorithm which allows the system to generate a new token at system initialization or after the token is lost (by the failure of some process).

Assume that the control token is lost, and that all active processes become aware of this fact (by some timeout mechanism). The algorithm must then generate exactly one control token within finite time. Since a process cannot know, in general, which other processes are active, assume that each process begins the algorithm knowing only its own (unique) identifier and the fact that it is in a ring. Also assume that each process executes at a finite, non-zero rate (the rate of each process is independent and the rates may vary with time), that no messages are lost and that all messages are delivered within a finite time after they are sent. Le Lann's algorithm, discussed below, also requires the assumption that messages are delivered in the same order that they were sent.

In Le Lann's algorithm an election, conduct*d as follows, is held to determine which process is to create the control token. Each process sends a message containing

its own identifier around the ring. It also records the identifiers from messages which it helps to send around the ring. When the process's own identifier returns, it checks to see whether it has the highest priority (based on an ordering of identifiers) among the active processes. If so, it creates a control token and stops the election.

Since every probe always goes all the way around the ring, Le Lann's algorithm always sends N^2 messages, where N is the number of active processes. (Note: we count a "message sent" every time a message passes between a pair of processes. Thus, a message which is forwarded all the way around the ring would count for N messages sent.) Chang and Roberts [CR77] show that this can be improved to N log N in the average case (assuming all possible permutations of priorities are equally likely) by modifying the algorithm so that higher priority processes do not retransmit the messages of lower priority processes (which could not win the election anyway). In the worst case, Chang and Roberts algorithm still sends $O(N^2)$ messages. (Note: For both Le Lann's and Chang and Robert's algorithms the number of messages sent depends only on the ordering of process identifiers around the ring. Hence, it is unnecessary to consider various interleavings when analyzing either of these algorithms.) Both of these algorithms send messages in only one direction.

Hirschberg and Sinclair [HS79] give an algorithm which solves the problem using only O(N log N) message passes in the worst case (messages may be sent in either direction around the ring). The bound is obtained by sending "probes" a fixed distance in each direction (the first probe is sent distance one, i.e., to its immediate neighbors). (A probe is conceptually a message which is passed from process to process around the ring.) After both probes are acknowledged with an indication that no process of a higher priority occurs within the given distance, the distance is doubled and another probe is sent. If a process with a higher priority than the probe sees it, a negative acknowledgment is returned, which causes the originating process to stop sending probes. Eventually, the highest priority process will send a probe which "wraps around" the ring. This process then creates a control token and takes control of the system. The algorithm uses no more than 16N + 8N log N messages, where N is the number of processes in the ring.

The next section introduces a model to describe message passing systems which are connected as rings. This is followed by a section defining Le Lann's problem in terms of the model. An improved version of Hirschberg and Sinclair's algorithm which sends no more than 4N + 6N log N messages (for N processes) is given next. The final

section proves that this solution is optimal to within a multiplicative constant; more than (1/8)N log (N/2) messages must be sent in the worst case for any solution with N processes.

Rings

A language has not yet been agreed upon with which to describe problems in distributed systems. A model will be given here for cyclically connected processes communicating by sending messages. A more general, and formal, model of distributed systems is given in Burns [Bur80].

A two-way process is a state transition system in which the possible transitions can be partitioned in left-sends, right-sends, left-receives and right-receives. Each process also has a designated state called the initial state of the process. (The details of the definition of the state transition system will be omitted. See Burns [Bur80] for a more complete treatment.) Every two-way process has a left and a right input queue. The value of an input queue is a sequence of messages, where a message is a element of an implied universal set of messages. A left-send causes a message to be appended to the right input queue of the process connected to the left of the sending process. The message sent and the next state of the sending process depend (deterministically) only on the state of the process. A right-send behaves symmetrically.

A left-receive attempts to receive a message which was previously sent to it by the process to its left (and so is in its left input queue). Right-receives are symmetric to left-receives. From the point of view of the process, this is a deterministic transition. A value is read from an external source; this value is either a message or a special null value indicating that no message is "ready". The choice of which message is chosen from the input (or whether any message is chosen) depends on the type of ring system chosen, as explained below. When a message is received, it is deleted from the input queue.

Note the somewhat subtle distinction between a receive transition and receiving a message. A process executes a receive transition whenever an attempt is made to receive a message, but it receives message only when this attempt is successful. There is no corresponding ambiguity with sends since they always succeed in adding a message to the appropriate input queue.

An N-ring is an N-tuple, R = (A1,A2,...,AN), of N two-way processes. An instantaneous description (id) of ring R = (A1,...,AN) consists of a state and two queues of message values for each process Ai, i $\in$ [N]. The initial id of R, initid(R), consists of the initial state of each process of R and the empty queue for each input queue of each process of R.

Three types of rings are defined, each of which handles the receive transition in a different way. We think of these as being different types of networks, since the definition of individual processes is unaffected. In a delay-free ring, a receive transition always chooses the oldest element from the input queue and chooses the null value if and only if the queue is empty. The lower bound result given below is for delay-free rings. In an ordered ring, a receive transition always chooses (non-deterministically) either the oldest element in the input queue or the null value. Le Lann's algorithm (discussed earlier) works for ordered rings. In an unrestricted ring, a receive transition may choose any element in the input queue or the null value. This models a system in which messages may be delayed arbitrarily and delivered in any order. The upper bound result given below works for unrestricted rings.

Computations for rings will be specified in a way similar to the way computations were specified for (M,N)-systems in Chapter II. However, in addition to specifying which process is to take the next step, the non-deterministic choice involved in receiving messages (in ordered and unrestricted rings) must also be resolved. Therefore, the components of a schedule of a ring must designate which process is to take the next step and, for

ordered rings, decide whether to choose a message or the null value for a receive transition. In an unrestricted ring, a choice of which message to receive must also be made. The details of specifying schedules of rings are omitted. However, we will assume that processes are represented by name, rather than position, and that the selection of which message to receive (if needed) is made by position in the input queue. (For example, a non-negative integer might be used to specify the choice. A value of zero or greater than the length of the queue would select "no message ready".)

Let q be an id and h be a schedule of ring R. Borrowing the notation of Chapter II, we let comp(q,h) designate the computation of R beginning at id q which is specified by schedule h. (The definition of schedules could vary for different types of rings. We will assume that the type of ring is understood and that an appropriate form for schedules is chosen.) Also, if h is finite, let r(q,h) designate the final id of R occurring in comp(q,h). Define msgs(q,h) to be the number of send transitions which occur in the computation from q by h. Define MSGS(R) = max {msgs(initid(R),h) : h is a schedule of R}.

We will only consider computations in which all processes continue to function and in which no message is left forever in an input queue while a process attempts to

receive it. A schedule h of ring R is fair from id q if every process of R takes an infinite number of steps in comp(q,h) and if every message in an input queue for which there are an infinite number of corresponding receive transitions in comp(q,h) is received in comp(q,h). In a delay-free ring, any schedule in which no process halts is automatically fair. Note that it is possible for a message to remain unreceived if the process which is to receive it only tries (executes the appropriate type of receive) a finite number of times.

The Election Problem

An election process is a process with a distinguished subset of states called election states. Let R = (A1,..., AN) be a ring composed of election processes. R is said to solve the election problem if for every schedule, h, of R which is fair from initid(R), there is one and only one i ∈ [N] such that Ai reaches an elected state in comp(initid(R),h).

The above definition captures the idea of electing a process in a particular ring, but Le Lann's problem requires that the election work for any arrangement of processes. In addition, the processes are not supposed to have prior knowledge of the number of other processes in the ring or of their identity. These conditions are described by the next definition.

If P is a set of two-way processes and R = (A1,..., AN) is a ring such that Ai ∈ P for each i ∈ [N] and such that all the Ai are distinct, then R is chosen from P. Let P be a countably infinite set of two-way election processes. If every ring R chosen from P solves the election problem, then P solves the general election problem.

The Algorithm

Figure 6-1 defines an infinite set of processes, one for each integer. The process with priority value I is called "process_I". The notation of the algorithm should be clear except, perhaps, for the instruction "send" and the function "receive". The "send(dir,msg)" instruction causes a message with the value in variable "msg" to be sent to the input queue in the direction given by "dir". The Boolean valued function "receive(dir,msg)" causes a receive to be attempted from the input queue in direction "dir". If the attempt is successful, variable "msg" is set to the value of the message, and the function has value "true". If no message is received, the function has value "false". It can thus be seen that "send" and "receive" correspond to the send and receive transitions of the model. The "elected" states of each process are exactly those in which variable "elected" is true.

```
program process_I;  (* I is unique for each process *)
   type direction = (left, right);
        message = record
                    mpri  : integer;
                    mdist : integer
                  end;
   var maxpri, dist : integer;
       elected, ack : Boolean;
       msg, mmsg    : message;
       dir, mdir    : direction;
   function rev( dir : direction): direction;
      begin
         if dir = left then rev := right
         else  rev := left
         end;

   begin
      maxpri    := I;
      msg.mpri  := I;
      msg.mdist := 1;
      dir       := left;
      elected   := false;
      while true do begin
         send(dir, msg); (* send probe *)
         ack := false;
         while not ack do
            for mdir := left to right do
               if receive(mdir,mmsg) then begin
                  if mmsg.mpri >= maxpri then with mmsg do
                     if mdist=0 then (* received ack *)
                        if mpri=I then ack := true
                        else send(mdir,mmsg)
                     else begin  (* received probe *)
                        maxpri :=mpri;
                        mdist := mdist -1;
                        if mpri=I then elected := true
                        else if mdist > 0 then
                           send(mdir,mmsg)
                           else send(rev(mdir),mmsg)
                        end
                  end;
         msg.mdist := 2*msg.mdist;
         dir       := rev(dir);
         end;
   end.
```

Solution to the General Election Problem

Figure 6-1

The algorithm solves the general election problem for unrestricted rings. When a ring is formed from any subset of the above defined processes, it will execute as follows (from the initial id). Each process tries to become elected by sending probes around the ring. (A probe contains the priority of the originating process and the remaining distance that it is to be sent.) A probe goes a fixed distance around the ring, and an acknowledgment is sent back if it does not encounter a priority with a higher value. (An acknowledgment is represented by a message with a distance field of zero.) Each successive probe sent by the same process goes twice as far as the last and moves in the opposite direction. The probes of the process with the highest priority value will always be acknowledged (in a fair schedule), and the acknowledgment will always return to the originating process. One of its probes will therefore eventually go all the way around the ring and reach the originating process, which causes the highest priority process to become elected. (Election occurs when a probe is received by its originating process.) Also, no other process can become elected since the highest priority process will not pass on messages from lower priority processes. The solution is thus easily seen to be correct in that it elects exactly one process in finite time.

Notice that the solution is designed specifically to solve the general election problem as formally specified. Details necessary for a more generally useful solution have been omitted. For example, additional types of messages are needed to terminate the election in order to do further useful processing.

The given solution is very similar to the previously described Hirschberg and Sinclair algorithm. The main differences are that probes are sent for a given distance in only one direction, which alternates on successive probes, and there is no negative acknowledgment for processes which cannot win the election (that is, probes of losing processes are "swallowed"). These modifications provide an algorithm which has better worst case performance than Hirschberg and Sinclair for any given arrangement of processes, although the order of Hirschberg and Sinclair's result has not been improved.

Theorem 6.1

There exists a solution, P, to the general election problem such that for any positive integer N and any unrestricted N-ring, R, chosen from P, MSGS(R) $\leq$ 4N + 6N log N.

Proof: Let P = {process_I : I is an integer}. Let R = (A1,...,AN) be any N-ring chosen from P, and let pri_1=I, where Ai = process_I for i $\in$ [N]. It has been

argued that R solves the election problem. It will now be demonstrated that R will use no more than 4N + 6N log N send transitions in doing so.

In order to compute the number of messages sent, all messages caused by probes which are sent with the same initial distance value by the various processes are grouped together. Probe-set i consists of all probes which are originated with a distance field of 2^i. Thus, the first probes sent by each process are part of probe-set 0. The processes which receive an acknowledgment of their first probe will send probes in probe-set 1, etc. Let S(i) be the set of processes which can send probes (in any computation for ring R) in the ith probe-set and let s(i) = |S(i)|, (initially, all processes send probes, so s(0)=N). Note that probes from different probe-sets may overlap in time.

Claim: The following inequality holds.

$$s(i) < N/(2^{i-2}) \quad \text{for } i > 1 \qquad (6.1)$$

Suppose there are two processes, q and r, in S(i) such that there are fewer than 2^{i-2} processes between q and r in R. But q and r have each sent successful probes distances of at least 2^{i-2} in each direction in order to be in S(i). But this implies that pri(q) > pri(r) and pri(r) > pri(q), which is impossible. Therefore, there are at least 2^{i-2} processes between every pair of processes in s(i).

This implies that $N \geq s(i)*(2^{-2}+1)$, so the claim holds.

We charge each probe-set with all the messages in sending the probes and returning acknowledgments of that set. Every probe in probe set i which is acknowledged accounts for $2*2^i$ messages. A probe which is not acknowledged can cause at most 2^i messages to be sent. Thus, the total number of messages sent for probe-set i is given by Cost(i) as follows:

$$\begin{aligned} \text{Cost}(i) &\leq 2*(2^i)*s(i+1) + (2^i)*(s(i) - s(i+1)) \\ &= s(i)*2^i + s(i+1)*2^i \end{aligned}$$

Let k = ceiling(log N). Note that S(i) is empty for i greater than k, so Cost(i) = 0 for i > k. The total number of messages sent, Total, can now be bounded as follows.

$$\begin{aligned} \text{Total} &= \text{Cost}(0) + \text{Cost}(1) + \ldots + \text{Cost}(k) \\ &\leq s(0)*2^0 + s(1)*2^0 + s(1)*2^1 + s(2)*2^1 \\ &\quad + \ldots + s(k-1)*2^{k-1} + s(k)*2^{k-1} \\ &\quad + s(k)*2^k \\ &\leq s(0) + 3*s(1)*2^0 + 3*s(2)*2^1 + \ldots \end{aligned}$$

Now substitute N for s(0) and s(1), and $N/(2^{i-2})$ for s(i) for i>1, by the claim.

$$\begin{aligned} \text{Total} &\leq N + 3*N + 3*[N*2^1/(2^0) + \ldots \\ &\quad + N*2^{k-1}/(2^{k-2})] \\ &\leq 4*N + 3*[2*N + 2*N + \ldots + 2*N] \\ &\leq 4*N + 3*[2*N]*(k-1) \end{aligned}$$

$$\leq 4*N + 6*N*(\log N)$$

This completes the proof of the theorem. □

The preceding analysis is conservative for N much smaller than the next higher power of two. For a power of two, we get the following, sharper result.

Theorem 6.2

There exists a solution, P, to the general election problem such that for any positive integer N which is a power of two and any N-ring chosen from P, MSGS(R) ≤ N + 3N log N.

Proof: Let N be any positive integer which is a power of two and let P and R be as in the proof of Theorem 6.1. The claim of the theorem can now be strengthened to

$$s(i) \leq N/(2^{i-1}) = 2^{k-i+1} \quad \text{for } i>1 \qquad (6.2)$$

since s(i) is integral. Substituting this tighter value into the formula for Total gives:

$$\begin{aligned} \text{Total} &\leq s(0) + 3*s(1)*2^0 + 3*s(2)*2^1 + \ldots \\ &\quad + 3*s(k)*2^{k-1} \\ &\leq N + 3*N + 3*[(2^{k-2+1})*2^1 + \ldots \\ &\quad + (2^{k-k+1})*2^{k-1}] \\ &\leq 4*N + 3*[2^k]*(k-1) \\ &\leq 4*N + 3*N*(\log N) - 3*N \\ &\leq N + 3*N*(\log N) \end{aligned}$$

This completes the proof. □

We conjecture that the actual worst case number of messages sent for arbitrary N is bounded by N + 3N log N. This is supported, by not proved, by the following analysis.

Let P = {process_I : I is an integer}, and let R be any N-ring chosen from P. Let h(N) be the number of messages which will be caused in R by the highest priority process in R. (A message is caused by a process if it results from a probe which was originated by the process.) We write h as a function of N rather than R since the value of h(N) is the same for all N-rings chosen from P. Clearly, h(1) = 1. For N>1, the highest priority process will send probes of distances 2^0, 2^1, 2^2,..., $2^{\lfloor \log (N-1) \rfloor}$ (each of which will be acknowledged), and a final probe which will go all the way around the ring. Therefore,

$$\begin{aligned} h(N) &= 2*(2^0 + 2^1 + \ldots + 2^{\lfloor \log (N-1) \rfloor}) + N \\ &= 2*(2^{\lfloor \log (N-1) \rfloor + 1} - 1) + N \\ &= 4*2^{\lfloor \log (N-1) \rfloor} + N - 2 \end{aligned}$$

The maximum number of messages sent in R by any process other than the process with the highest priority is determined only by how close the process is to processes with higher priority. Let p be a process of R which is does not have the highest priority. The process which is the closest to p on p's left and which has a higher

priority than p is the left boundary of p. The similar process to p's right around the ring is the right boundary of p. The part of the ring which is between the left boundary of p and the right boundary of p (including p but not either boundary) is the segment of p. The length of a segment is the number of processes that it contains. Let f(n) be the maximum number of messages sent in any segment of length n in any ring chosen from P. Since the segment of the second highest priority process in R contains every process other than the highest priority process in R, it should be clear that the number of messages sent in a computation of R is bounded above by h(N) + f(N-1).

The position of p (in p's segment) is the number of processes in the segment of p which occur to the left of p. Thus, the position of p can range from zero to one less than the length of p's segment. Let g(k,n) be the maximum number of messages which may be caused by a process in a ring chosen from P which has a segment of length n and is in position k. (Note that the value of g depends only on the length of the segment and the position of the process within it, not on the particular processes making up the segment.)

The algorithm send probes to the left with increasing distances of even powers of two (starting with 2^0) and to the right with increasing distances of odd powers of two.

For k>0, let lmax(k) = 2* $\lfloor \log_4(k) \rfloor$, and let rmax(k) = 2* $\lfloor \log_4(2k) \rfloor$ - 1. Also let lmax(0) = 0 and rmax(0) = -1. For k>0, lmax(k) and rmax(k) give the maximum even and odd integers, respectively, whose power of two is less than or equal to k. The power of two of the last successful probe which may be sent by a process in position k of its segment of length n is then given by probes(k,n) = min[lmax(k), rmax(n-k-1)] + 1.

For n>0, g(0,n) = 1, since the first probe is sent to the left and swallowed. The value of g(k,n) for n>k>0 is given below. Let i = probes(k,n). Let process p be in position k of its segment which has length n. The first i probes sent by p will be acknowledged and will therefore account for $2*(2^0 + 2^1 + \ldots + 2^i) = 2^{i+2} - 2$ messages sent. The number of messages sent by the last probe (which will be swallowed when it reaches the boundary of the segment) depends on whether it is going right (i is even) or left (i is odd).

If i is even, then $g(k,n) = 2^{i+2} + n - k - 2$.

If i is odd, then $g(k,n) = 2^{i+2} + k - 1$.

Directly from the definitions we obtain the following.

$f(0) = 0$

$$f(n) = \max_{0 \le k < n} (f(k) + g(k,n) + f(n-k-1))$$

Using this definition and the definition of g(k,n),

successive values of f(N) can easily be computed. This has been done for every positive N less than 3000, and in all cases $h(N) + f(N-1) \le N + 3N \log N$. However, all attempts have failed to show that the conjecture is true for all N.

The Lower Bound

If a ring R has an even number of processes, R = (p1,p2,...,p2N), then the center process of R is pN. Let a1,...,aA and b1,...,bB be distinct two-way processes and let R1 = (a1,...,aA) and R2 = (b1,...,bB). Define join(R1,R2) to be (a1,...,aA,b1,...,bB). The join operator "pastes together" two rings. For rings R1, R2 and R3, let join(R1,R2,R3) = join(join(R1,R2), R3).

A schedule h of a ring R=(A1,...,AN) which has the property that process AN does not have any steps is a joining schedule of R. (Joining schedules are used for rings which are to be "broken apart" and combined with other rings.) An id q of R is called quiescent if for every schedule h of R, msgs(q,h) = 0. An id q of R is called join-quiescent if for every joining schedule h of R, msgs(q,h) = 0.

Note that if h_1 is a joining schedule of R1 and h_2 is a joining schedule of R2, then h_1h_2 and h_2h_1 are joining schedules of join(R1,R2). Also, there can be no interaction between the processes of R1 and R2 in ring join(R1,R2) under schedule h_1h_2 or h_2h_1.

Let L be a set of rings chosen from a set of two-way processes, P. L is <u>compatible</u> if for every R1 = (A1,...,Aa) and R2 = (B1,...,Bb) in L, the sets {A1,...,Aa} and {B1,...,Bb} are disjoint.

<u>Lemma 6.1</u>

Let P be a solution to the general election problem. For every i>0 there is an infinite compatible set, L_i, of delay-free 2^i-rings chosen from P such that for every R $\in L_i$, there is a finite joining schedule h such that msgs(initid(R),h) > (1/4)N log N, where $N = 2^i$.

<u>Proof</u>: By induction on i.

BASIS. i=1. We must find an infinite set of 2-rings which will send at least one > (1/4)*2 log 2 = 1/2 messages. Suppose there are two processes p and p' in P which will not send a message unless they first receive a message. Then p and p' will each become elected in rings (p) and (p') without sending any messages (since P solves the general election problem). But then p and p' can both become elected in ring (p,p'), a contradiction. Therefore, there can be at most one process in P which will not send a message before receiving one. Let P' and P" be infinite, disjoint subsets of P which do not contain this (possibly existing) process. For any proceses p' $\in$ P' and p" $\in$ P", there must be a joining schedule of ring (p',p") which causes p' to send a messages. Therefore, the lemma holds

for i=1 with $L_1 = \{ (p',p") : p' \in P' \text{ and } p" \in P" \}$.

INDUCTIVE STEP. Assume that the lemma is true for i-1. Let $N = 2^i$. Let R, R' and R" be any three rings in L_{i-1}.

<u>Claim</u>: Let Ra = join(R,R'), Rb = join(R',R"), Rc = join(R",R), Rd = join(R',R), Re = join(R",R') and Rf = join(R,R"). For some ring Rx $\in$ {Ra,Rb,Rc,Rd,Re,Rf}, there exists a joining schedule s of Rx such that msgs(initid(Rx),s) > (1/4)N log N messages.

Since sets of three rings can repeatedly be chosen from L_{i-1} without repetition, it is clear that the claim implies the inductive step.

We now show the claim. By the inductive assumption, there must be a finite joining schedule, h, of R for which msgs(initid(R),h) $\geq$ (1/4)(N/2) log (N/2). Also, we may assume that r(initid(R),h) is join-quiescent, since, if not, we may extend h until either a join-quiescent state is reached or until more than (1/4)N log N messages are sent, in which case the claim holds trivially for Ra. Let h' and h" be similar joining schedules for R' and R", respectively.

Let q_a = r(initid(Ra),hh'). By assumption, r(initid(R),h) and r(initid(R'),h') are join-quiescent. Note that hh' is a joining schedule of Ra. Let s be any joining schedule of Ra from q_a (so hh's is a joining schedule of Ra from initid(Ra)). The first process to send

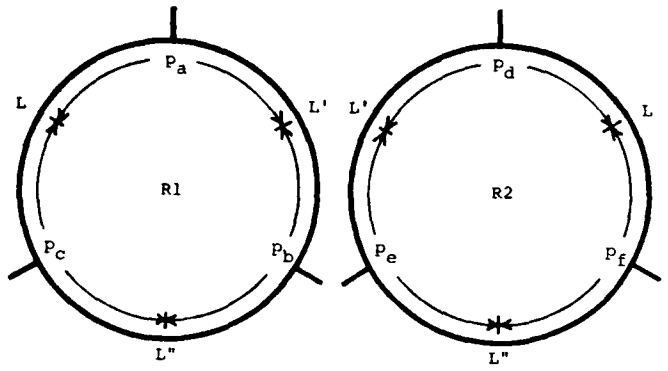
a message in comp(q_a,s) must be the center process of Ra (i.e., the rightmost process of R). (Otherwise, we could find a joining schedule of R or R' extending h or h' which would send an additional message in R or R', respectively, contrary to assumption.) The second message sent in comp(q_a,s) must either come from the center process of Ra or must come from the receiver of the first message. It is easy to see that the set of all processes sending messages in comp(q_a,s) are contiguous in Ra and include the center process of Ra. This fact is used below.

Let P_a be the N/2 - 1 processes of Ra which are less than distance N/4 from the center process of Ra. (That is, P_a consists of the processes between (but not including) the center processes of R and R'.) If an unbounded number of messages can be sent from q_a in Ra for a joining schedule of Ra the claim holds, so let h_a be a finite schedule of Ra consisting only of steps of the processes in P_a such that no extension of h_a consisting of steps of P_a will cause a message to be sent in comp(q_a,h_a).

Suppose the claim is false. Then we must have msg(q_a,h_a) < N/4 because msgs(initid(Ra),hh'h_a) = msgs(initid(R),h) + msgs(initid(R'),h') + msgs(q_a,h_a) > 2*((1/4)N/2 log (N/2)) + msgs(q_a,h_a) = (1/4)N log N - N/4 + msgs(q_a,h_a). This implies that less than N/4 processes send messages during comp(q_a,h_a). By the argument given earlier, these processes are contiguous in Ra and include the center process of Ra. Therefore, no message is sent to a process outside of P_a during comp(q_a,h_a). We say that all messages sent in this computation are <u>local</u> to P_a.

Define $h_b,\ldots,h_f$ and $P_b,\ldots,P_f$ in a similar way to h_a and P_a for Rb,...,Rf, respectively. Note that P_a, P_b and P_c are mutually disjoint, as are P_d, P_e and P_f.

Let Rl = (join(R,R',R")) and R2 = (join(R,R",R')) (see Figure 6-2). Consider q_1 = r(initid(Rl), hh'h"$h_a h_b h_c$). The only processes which can send messages from r(initid(Rl),hh'h") are the rightmost processes of R, R' and R" (by the choice of h, h' and h"). Since P_a, P_b and P_c are mutually disjoint, the message sending activity is local to each set during comp(initid(Rl),hh'h"$h_a h_b h_c$). Therefore, the processes in P_a and their input queues are in the same state at id q_1 in Rl as they are at id q_a in Ra, and similar statements hold for P_b and P_c. Thus, all messages which will be sent because of the joins have already been accounted for in h_a, h_b and h_c. This implies that q_1 and (by a similar argument) q_2 = r(initid(R2), hh'h"$h_d h_e h_f$) are quiescent.



Representation of Rings R1 and R2

Figure 6-2

Since q_1 and q_2 are quiescent, for each ring, there are processes, p_1 and p_2, which will be elected on their own (with no further messages sent) from q_1 and q_2, respectively. That is, there is a finite (possibly empty) schedule h_1 specifying that only p_1 is to take steps such that process p_1 is elected at $r(q_1,h_1)$. (Note: it may not be apparent that p_1 and h_1 exist since h_1 is not fair. But clearly, some process must be elected for any fair schedule. Since no new messages are sent after q_1, the behavior of the elected process will be the same if the steps of all the other processes are removed from the computation.) A similar schedule, h_2, exists for p_2.

Assume, without loss of generality, that $p_1 \in P_a$. There are three cases for p_2.

Case 1: $p_2 \in P_d$. Note that Ra is composed exactly of the processes in P_a and P_d. Moreover, the processes in each of these sets have exactly the same neighbors in Ra as in R1 and R2, respectively. Therefore, the behavior of processes in P_a and P_d must be the same in Ra from $r(initid(Ra),hh'h_ah_d)$ as in R1 from q_1 and in R2 from q_2, respectively. In particular, p_1 and p_2 are both elected in Ra at id $r(initid(Ra), hh'h_ah_dh_1h_2)$, which contradicts the hypothesis of the lemma.

Case 2: $p_2 \in P_e$. The processes in P_c and P_f must behave the same in Rc from $r(initid(Rc),hh"h_ch_f)$ as in R1 from q_1 and R2 from q_2, respectively. But then no process can be elected in Rc, another contradiction.

Case 3: $p_2 \in P_f$. This case is symmetric to Case 2 since Rb consists of exactly those processes in P_b and P_e.

The assumption that the claim is false leads to contradiction, hence the claim is proved and the lemma holds. □

Theorem 6.3

If P is a solution to the general election problem, then for all $N \geq 1$, there exists a delay-free N-ring, R, chosen from P such that $MSGS(R) > (1/8)N \log (N/2)$.

Proof: Let i = floor(log N). By the lemma, a 2^i-ring Rx of P and a joining schedule s of Rx can be found such that msgs(initid(Rx),s) > $(1/4)*2^i*(\log 2^i)$ = $(1/8)*2^{i+1}*(\log 2^{i+1}/2) \geq (1/8)*N*(\log (N/2))$. Such a ring can easily be incorporated in a ring of P of length N, so the theorem is true. □

Theorem 6.4

If N is a power of 2 and P is a solution to the general election problem, then there is a delay-free N-ring, R, chosen from P such that MSGS(R) > (1/4) N log N.

Proof: This follows directly from Lemma 6.1. □

CHAPTER VII

SUGGESTIONS FOR FURTHER WORK

The study of parallel computing systems is a relatively new and rapidly growing area. This thesis is concerned with theoretical aspects of parallel systems in which processes are allowed to run completely asynchronously. The main object of study is the cost of coordinating the actions of many independent processes, where cost is measured by the size of the shared variables or the number of messages sent, depending on the communication model being used.

This thesis examines algorithms using reads and writes or test-and-sets for communicating through shared variables. Many other communication operations can be defined which are intermediate in power between reads and writes and the test-and-set operation. For example, Friedman and Wise [FW78] have proposed the "sting" operation. A sting can write a variable with a value which is a function of the variable's current value (as the test-and-set does), but no information is returned to the process executing the sting. A separate read of the variable must be made to determine the result of the sting operation. Friedman and Wise argue that the sting

operation can be implemented to be faster and possibly cheaper that the test-and-set operation. By using techniques developed in this thesis, a measure of the complexity of solving a particular problem with a particular communication operation may be found. This may give a way to judge the relative merits of different operations for particular applications.

The work in Chapters III, IV and V primarily deals with a complexity measure (amount of shared space) which is analogous to the space complexity measure for sequential systems. A complexity measure analogous to time in sequential systems would also be of interest. Since time is explicitly omitted from the model used, this presents some difficulties.

First, since the problems studied do not generally have fixed starting and stopping points, it is necessary to define a segment of a computation to be used in calculating such a measure. For example, in a mutual exclusion problem we might choose a segment beginning when a particular process leaves its remainder region and ending when that process returns to its remainder region (if this occurs).

Second, some way of measuring the elapsed time in a chosen segment must be selected. One obvious measure is to count the total number of steps which are taken in the segment or to count the steps of a distinguished process

(probably the one that was used to define the segment). This measure may be useful if we are interested in average case behavior, but it is inadequate for worst case behavior because we can always pack an unbounded number of steps into a segment whenever a process reaches a point at which it must wait. A possible way around this difficulty is just to not count the steps of any process which is in a wait loop, but this seems somewhat unnatural.

A measure (suggested by Lamport [Lam77d]) which is more appealing intuitively is the "slow clock" measure. At any time during a computation, some process has the slowest clock. Assume that a clock pulse occurs at the beginning of the segment. Clock pulses are then calculated by keeping track of which processes have taken at least one step since the previous clock pulse. Another clock pulse occurs as soon as every process which is active (not in a region where it is allowed to halt) has taken at least one step since the previous clock pulse. The slow clock measure may be modified to require every process to take at least k steps (for some constant k) before a clock pulse is counted.

Another measure uses a "timing", which assigns a time value (increasing) to each step of a computation. A timing is acceptable if it meets certain constraints (e.g., the time difference between any two steps of the same process

might be bounded by a constant). The worst case time for a particular computation is measured by the elapsed time according to the worst case timing. This type of measure was apparently first proposed by Peterson and Fischer [PF77] and has been refined and applied by Peterson [Pet79b] and Lynch [Lyn80]. Future work will investigate these measures and possibly others applied to various synchronization problems of asynchronous systems.

The results in Chapter VI give upper and lower bounds on the number of messages passed in solving the election problem in a ring network. The election problem has an obvious extension to other communication networks. Work is planned to investigate the election problem for general graphs.

This thesis has shown a few results about asynchronous systems of parallel processes. Perhaps more important, techniques have been developed for proving facts about very complicated objects. These tools should be useful for further theorectical investigations into asynchronous systems.

ACKNOWLEDGMENTS

Anyone who has undertaken the writing of a doctoral dissertation knows that many different people assist the author in many different ways. This acknowledgment gives thanks to some of those who have helped me over the last few years.

First I acknowledge the help and encouragement of my thesis committee: Nancy A. Lynch, Richard A. DeMillo, Lucio Chiaraviglio and Michael J. Fischer. Co-chairman Nancy Lynch worked closely with me on many of the results appearing in the thesis. She not only inspired me to search for new results, but her incisive criticism greatly helped me in producing readable and convincing proofs. Co-chairman Richard DeMillo gave me early support and encouragement and carefully read and criticized each draft of the thesis. Lucio Chiaraviglio first sparked my interest in theoretical computer science in his courses on recursive function theory. I also thank him for advice throughout my tenure as a graduate student and for carefully reading the thesis. My first work on the problems addressed in this thesis began as part of a seminar at Georgia Tech which was led by Nancy Lynch and Michael Fischer. Michael Fischer of the University of

Washington has provided stimulating ideas and criticism throughout the development of this thesis.

I also thank all the faculty in the School of Information and Computer Science at Georgia Tech. For especially stimulating discussions a special thanks goes to Albert N. Badre, Philip H. Enslow and James Gough, Jr. I am also grateful for the assistance of staff members Allen Akin, Ed Coleman, Perry Flinn and Dan Forsyth.

For their friendship, companionship and encouragement, I thank my fellow graduate students at Georgia Tech: Allen Acree, Elaine Strong Acree, Jack Corley, Edith Martin, Barney McCabe, Wayne McCoy, Michael Merritt, Lionel Rodriguez, Tim Saponas, Shelley Smith and Mark Turner.

The following faculty members at Indiana University have been kind enough to read and criticize certain parts of this thesis: Robert Filman, Daniel Friedman, Edward Robertson and Paul Purdom.

The results in Chapter VI grew directly out of a conversation with Dan Hirschberg of Rice University, and I thank him for providing me with a stimulating problem.

Many other individuals have been kind enough to comment on parts of the thesis while it was being developed. I especially wish to thank Leslie Lamport of SRI and Dick Lipton of Princeton University for their comments.

I owe a special debt of thanks to William H. Cotterman, faculty member at Georgia State University, who is primarily responsible for encouraging me to purse the degree of Doctor of Philosophy. I also thank Nuclear Assurance Corporation and President Paul F. Schutt for their assistance and encouragement.

I sincerely thank my wife, Judith, for her support, given so freely and in so many ways and my daughter, Mary Ellen, with whom I plan to spend much more time in the future.

Support for this thesis was provided in part by a Presidential Fellowship from the Georgia Institute of Technology, ONR grant N00014-79-C-0231, and NSF grants MCS77-15628 and MCS77-28305.

BIBLIOGRAPHY

AHU74 Aho, A., Hopcroft, J., and Ullman, J. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass. (1974).

And177 Andler, S. Synchronization primitives and the verification of concurrent programs. CMU-TR-77-106, Dept. of Comp. Sci., Carnegie-Mellon Univ., May 1977, 47 pp.

And76 Andrews, G.R. Message classes: an approach to process synchronization. TR 76-275, Dept. of Comp. Sci., Cornell Univ., Apr. 1975, 27 pp.

And79 Andrews, G.R. Synchronizing resources. Manuscript, Dept. of Comp. Sci., Cornell Univ., Feb. 1979, 56 pp.

Bab79 Babich, A.F. Proving total correctness of parallel programs. IEEE Transactions on Software Engineering SE-5, 6 (Nov. 1979), 558-574.

Bra78 Brand, D. Algebraic simulation between parallel programs. IBM Computer Science Research Report RC 7206, June 1978, 39 pp.

Bri72a Brinch Hansen, P. A comparison of two synchronizing concepts. Acta Informatica 1 (1972), 190-199.

Bri72b Brinch Hansen, P. Structured multi-programming. Comm. ACM 15, 7 (July 1972), 574-578.

Bri75 Brinch Hansen, P. The programming language concurrent pascal. IEEE Trans. on Software Engineering SE-1, 2 (Jun. 1975), 199-207.

Bur78 Burns, J.E. Mutual exclusion with linear waiting using binary shared variables. SIGACT News 10, 2, (Summer 1978), 42-47.

Bur80 Burns, J.E. A formal model for message passing systems. Comp. Sci. Dept. Tech. Rpt. 91, Indiana University, May 1980, 20 pp.

BL80 Burns, J.E., and Lynch, N.A. Mutual exclusion using indivisible reads and writes. Proc. 18th Annual Allerton Conf. on Communication, Control, and Computing, Oct. 1980.

BFJLP78 Burns, J.E., Fischer, M.J., Jackson, P., Lynch, N.A., and Peterson, G.L. Shared data requirements for implementation of mutual exclusion using a test-and-set primitive. Proc. of the 1978 International Conf. on Parallel Processing, Aug. 1978, pp. 79-87.

CLM76 Cardoza, E., Lipton, P.J., and Meyer, A.R. Exponential space complete problems for Petri nets and commutative subgroups. Proc. Eighth Annual ACM Symp. on the Theory of Computing, May 1976, pp. 50-54.

CP78 Case, R.P., and Padegs, A. Architecture of the IBM System/370. Comm. ACM 21, 1 (Jan. 1978), 73-96.

CR77 Chang, E., and Roberts, R. An improved algorithm for decentralized extrema-finding in circular configurations of processes. Comm. ACM 22, 5 (May 1977), 281-283.

CHP71 Courtois, P.J., Heyman, F., and Parnas, D.L. Concurrent control with "readers" and "writers". Comm. ACM 14, 10 (Oct. 1971), 667-668.

CHP72 Courtois, P.J., Heyman, F., and Parnas, D.L. Comments on "A comparison of two synchronizing concepts" by P.B. Hansen. Acta Informatica 1 (1972), 375-376.

CH75 Cremers, A., and Hibbard, T.N. An algebraic approach to concurrent programming control and related complexity problems. University of Southern California Computer Science Department technical report , Nov. 1975, 18 pp.

CH78 Cremers, A., and Hibbard, T.N. Mutual exclusion of N processors using an O(N)-valued message variable. (extended abstract). Lecture Notes in Computer Science 62, Springer-Verlag (Jul. 1978), 165-176.

CH79 Cremers, A., and Hibbard, T. Arbitration and queueing under limited shared storage requirements. Forschungsbericht Nr. 83, Universitat Dortmund, 1979, 14 pp.

deB67 de Bruijn, N.G. Additional comments on a problem in concurrent programming control. Comm. ACM 10, 3 (Mar. 1967), 137.

DLP79 DeMillo, R.A., Lipton, R.J., and Perlis, A.J. Social processes and proofs of theorems and programs. Comm. ACM 22, 5 (May 1979), 271-280.

DM79 DeMillo, R.A., and Miller, R.E. Implicit computation of synchronization primitives. Information Processing Letters 9, 1 (July 1979), 35-38.

Dij65 Dijkstra, E.W. Solution of a problem in concurrent programming control. Comm. ACM 9, 9 (Sep. 1965), 569.

Dij68a Dijkstra, E.W. Cooperating sequential processes. In Programming Languages, F. Genuys, (Ed.), Academic Press, New York, N.Y., (1968).

Dij68b Dijkstra, E.W. The structure of the "THE" multiprogramming system. Comm. ACM 11, 5 (May 1968), 341-347.

Dij71 Dijkstra, E.W. Hierarchical ordering of sequential processes. Acta Informatica 1 (1971), 115-138.

Dij72a Dijkstra, E.W. A class of allocation strategies inducing bounded delays only. AFIPS Conf. Proc., Vol. 40, 1972 SJCC, pp. 933-936.

Dij72b Dijkstra, E.W. Information streams sharing a finite buffer. Information Processing Letters 1 (1972), 179-180.

Dij74 Dijkstra, E.W. Self-stabilizing systems in spite of distributed control. Comm. ACM 17, 11 (Nov. 1974), 643-644.

Dij76 Dijkstra, E.W. A Discipline of Programming, Prentice-Hall (1976).

DLM78 Dijkstra, E.W., Lamport, L., Martin, A.J., Schoten, C.S., and Steffens, E.M.F. On-the-fly garbage collection: an exercise in cooperation. Comm. ACM 21, 11 (Nov. 1978), 966-975.

EiMc72 Eisenberg, M.A., and McGuire, M.R. Further comments on Dijkstra's concurrent programming control problem. Comm. ACM 15, 11 (Nov. 1972), 999.

ElMi79 Elgot, C.C., and Miller, R.E. On coordinated sequential processes. IBM Computer Science Research Report RC 7778, Aug. 1979, 47 pp.

Fel77 Feldman, J.A. Synchronizing distant cooperating processes. Tr 26, Dep. of Comp. Sci., Univ. of Rochester, Oct. 1977, 32 pp.

FLBB79 Fischer, M.J., Lynch, N.A., Burns, J.E., and Borodin, A. Resource allocation with immunity to limited process failure. Proc. 20th Annual Symp. on Foundations of Computer Science, Oct. 1979, pp. 234-254.

FW78 Friedman, D.P., and Wise, D.S. A conditional, interlock-free store instruction. Comp. Sci. Dept. Tech. Rpt. No. 74, Indiana University, Dec. 1978, 12 pp.

FW79 Friedman, D.P., and Wise, D.S. An approach to fair applicative multiprogramming. Tech. Rpt. No. 84, Indiana University, 1979, 23 pp.

Ger77 Ge[illegible], A.J. Process synchronization by counter variables. Operating System Reviews 11, 4, (Oct. 1977).

Goe77 Goeman, H.J.K. The arbiter: an active system component for implementing synchronizing primitives. Rpt. No. 77-4, Inst. of Applied Math. and Comp. Sci., Univ. of Leiden, Mar. 1977, 18 pp.

Gri77 Gries, D. An exercise in proving parallel programs correct. Comm. ACM 20, 12 (Oct. 1977), 921-930.

Hab69 Haberman, A.N. Prevention of system deadlocks. Comm. ACM 12, 7 (Jul. 1969), 373-377.

Hab72 Habermann, A.N. Synchronizing of commmunicating processes. Comm. ACM 15, 3 (Mar. 1972), 171-176.

HZ80 Henderson, P.B., and Zalcstein, Y. Synchronization problems solvable by generalized systems. J. ACM 27, 1 (Jan. 1980), 60-71.

HB77 Hewitt, C., and Baker, H. Laws for communicating parallel processes. Information Processing 77, Gilchrist, B., (Ed.), North-Holland Publishing Company (1977), 987-992.

HS79 Hirschberg, D.S., and Sinclair, J.B. An efficient algorithm for decentralized extrema-finding in circular configurations of processors. Manuscript, Rice University, 1979, 5 pp.

Hoa72 Hoare, C.A.R. Towards a theory of parallel programming. In Operating Systems Techniques, Hoare, C.A.R., and Perott, (Eds.), Academic Press, New York, 1972.

Hoa78a Hoare, C.A.R. Communicating sequential processes. Comm. ACM 21, 8 (Aug. 1978), 666-677.

Hoa78b Hoare, C.A.R. Corrigendum: communicating sequential processes. Comm. ACM 21, 11 (Nov. 1978), 958.

HC70 Holt, A., and Commoner, F. Events and conditions. Project MAC Conf. on Concurrent Systems and Parallel Computation, June 1970, pp. 3-52.

Hol71 Holt, R.C. Comments on prevention of system deadlocks. Comm. ACM 14, 1 (Jan. 1971), 36-38.

How76 Howard, J.H. Proving monitors. Comm. ACM 19, 5 (May 1976), 273-279.

How78 Howard, J.H. Proof of a semaphore primitive. IBM Computer Science Research Report RJ 2303, Aug. 1978, 13 pp.

KarM69 Karp, R.M., and Miller, R.E. Parallel program schemata. J. Comp. and Sys. Sciences 3, (1969), 147-195.

KasM79 Kasai, T.K., and Miller, R.E. Homomorphisms between models of parallel computation. IBM Computer Science Research Report RC 7796, Aug. 1979, 72 pp.

Kat78 Katseff, H.P. A new solution fo the critical section problem. Proc. Tenth Annual ACM Symp. on Theory of Computing, May 1978, pp. 86-88.

Kel74 Keller, R.M. A fundamental theorem of asynchronous parallel compuation. Lecture Notes in Computer Science 24, Parallel Processing, Goos, G., and Hartmanis, J., (Eds.), Springer-Verlag, New York (Aug. 1974), 102-112.

Kel76 Keller, R.M. Formal verification of parallel programs. Comm. ACM 19, 7 (July 1976), 371-384.

Kel78 Keller, R.M. Sentinels: a concept for multiprocess coordination. Technical report UUCS-78-104, Dept. of Comp. Sci., University of Utah, June 1978, 30 pp.

Knu66 Knuth, D.E. Additional comments on a problem in concurrent control. Comm. ACM 9, 5 (May 1966), 321-322.

Kos73 Kosaraju, R.S. Limitations of Dijkstra's semaphore primitives and Petri nets. ACM SIGOPS (Oct. 1973), 122-126.

Lad79 Ladner, R.E. The complexity of problems in systems of communicating sequential processes. Proc. 11th ACM Symp. on Theory of Computing, April 1979, pp. 214-223.

Lam74 Lamport, L. A new solution of Dijkstra's concurrent programming problem. Comm. ACM 17, 8 (Aug. 1974), 453-455.

Lam75a Lamport, L. Formal correctness proofs for multiprocess algorithms. Massachusetts Computer Assoc. manuscript, Oct. 1975, 13 pp.

Lam75b Lamport, L. Garbage collection by multiple processes: an exercise in parallelism. Massachusetts Computer Associates report CA-7507-1011, July 1975, 8 pp.

Lam76a Lamport, L. Time, clocks and the ordering of events in a distributed system. Mass. Comp. Assoc. Report CA-7603-2911, Mar. 1976, 23 pp.

Lam76b Lamport, L. Towards a theory of correctness for multi-user data base systems. Mass. Comp. Assoc. Report CA-7610-0712, Oct. 1976, 25 pp.

Lam77a Lamport, L. Proving correctness of multiprocess programs. IEEE Transactions on Software Engineering SE-3, 2 (Mar. 1977), 125-143.

Lam77b Lamport, L. Concurrent reading and writing. Comm. ACM 20, 11 (Nov. 1977), 806-811.

Lam77c Lamport, L. A new approach to proving the correctness of multiprocess programs. Manuscript No. 43, SRI International, Oct. 1977, 25 pp.

Lam 77d Lamport, L. Personal communication.

Lam78 Lamport, L. The implemenation of reliable distributed multiprocess systems. Computer Networks 2 (1978), 95-114.

LC75 Lauer, P.E., and Campbell, R.H. Formal semantics of a class of high-level primitives for coordinating concurrent processes. Acta Informatica 5, (1975), 297-332.

LeL77 LeLann, G. Distributed systems - towards a formal approach. Information Processing 77, Gilchrist, B., (Ed.), North-Holland Publishing Company (1977), 155-160.

Lip73 Lipton, R.J. On synchronization primitive systems. Yale research report #22, Oct. 1973, 109 pp.

Lip74a Lipton, R.J. Limitations of synchronization primitives with conditional branching and global variables. Proc. Sixth Annual Symp. on Theory of Computing, 1974.

Lip74b Lipton, R.J. Reduction: a method of proving properties of parallel programs. Comm. ACM 18, 12 (Dec. 1975), 717-721.

LZS74 Lipton, R.J., Zalcstein, Y., and Synder, L. A comparative study of models of parallel computation. Proc. 15th Annual Symp. on Switching and Automata Theory, 1974.

Lyn80 Lynch, N.A. Fast allocation of nearby resources in a distributed system. Proc. 12th ACM Symp. on Theory of Computing, April 1980, pp. 70-81.

LF79 Lynch, N.A., and Fischer, M.J. On describing the behavior and implementation of distributed systems. Lecture Notes in Computer Science 70, Semantics of Concurrent Computation, Goos, G., and Hartmanis, J., (Eds.), Springer-Verlag (1979), 147-171.

Mil173 Miller, R.E. A comparison of some theoretical models of parallel computation. IEEE Transaction on Computers C-22, 8 (Aug. 1973), 710-717.

Mil176 Miller, R.E. Talk notes on Mathematical studies of parallel computation. Presented at IBM Japan Symp. on Mathematical Foundations of Computer Science, Oct. 1976, 14 pp.

Mil177 Miller, R.E. Theoretical studies of asynchronous and parallel processing. Proc. of the 1977 Conf. on Information Sciences and Systems, Mar. 1977, pp. 333-339.

MY77 Miller, R.E., and Yap, C. Formal specification and analysis of loosely connected processes. IBM Research Report RC 6716, Sep. 1977.

MY78 Miller, R.E., and Yap, C. On formulating simultaneity for studying parallelism and synchronization. Proc. 10th ACM Symp. on Theory of Computing, (May 1978), pp. 105-113.

MN79 Milne, G., and Milner, R. Concurrent processes and their syntax. J. ACM 26, 2 (Apr. 1979), 302-321.

Miz78 Mizell, D. Verification and design aspects of "true" concurrency. Conf. Record 5th Annual ACM Symp. on Principles of Programming Languages, Jan. 1978, pp. 171-175.

Mor79 Morris, J.M. A starvation-free solution to the mutual exclusion problem. Information Processing Letters 8, 2 (Feb. 1979), 76-80.

Mul78 Muldner, T. On the synchronization of parallel computations. Simula Newsletter 6, 1 (Feb. 1978), 9-10.

Owi76 Owicki, S. A consistent and complete deductive system for the verification of parallel programs. Proc. 8th ACM Symp. on the Theory of Computing, May 1976, pp. 73-86.

Par75 Parnas, D.L. On a solution to the cigarette smokers' problem (without conditional statements). Comm. ACM 18, 3 (Mar. 1975), 181-183.

Pat71 Patil, S.S. Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes. Project MAC Computational Structures Group Memo 57, Feb. 1971.

Pet79a Peterson, G.L. Time-space trade-offs for asynchronous parallel models. Proc. 11th ACM Symp. on Theory of Computing, April 1979, pp. 224-230.

Pet79b Peterson, G.L. Concurency and complexity. TR59, Univ. of Rochester, Comp. Sci. Dept., Aug. 1979, 39 pp.

Pet80 Peterson, G.L. New bounds on mutual exclusion problems. TR68, Univ. of Rochester, Comp. Sci. Dept., Feb. 1989, 39 pp.

PF77 Peterson, G.L., and Fischer, M.J. Economical solutions for the critical section problem in a distributed system. Proc. 9th ACM Symp. on Theory of Computing, 1977, pp. 91-97.

Pip78 Pippenger, N. Private communication. Mar. 1978.

Rei78 Reif, J.H. Analysis of communicating processes. TR30, Comp. Sci. Dept., Univ. of Rochester, May 1978, 45 pp.

RP76 Rivest, R.L., and Pratt, V.R. The mutual exclusion problem for unreliable processes: preliminary report. Proc. 17th Annual Symp. on Foundations of Computer Science, Oct. 1976, pp. 1-8.

RV77 Robert, P., and Verjus, J. Toward autonomous descriptions of synchronization modules. Information Processing 77, Gilchrist, B. (Ed.), North-Holland (1977), 981-986.

Sha74 Shastry, S.K. A control structure for parallel processing. Lecture Notes in Computer Science 24, Parallel Processing, Goos, G., and Hartmanis, J., Eds., Springer-Verlag, New York (Aug. 1974), 132-147.

Ton76 Tonge, F.M. Expressions for time and space in a recursive realization of parallelism. U.C. Irvine, Tech. Rep. No. 79, 1976, 28 pp.

VvL72 Vantilborgh, H., and van Lamsweerde, A. On an extension of Dijkstra's semaphore primitives. Information Processing Letters 1 (1972), 181-186.

Wil78 Wileden, J.C. Modelling parallel systems with dynamic structure. COINS Tech. Rpt. 78-4, Univ. of Mass., Jan. 1978, 247 pp.

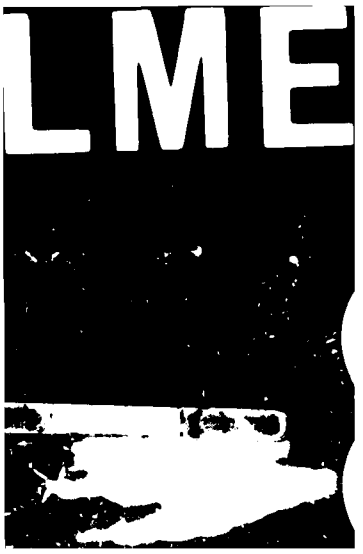
Wir77 Wirth, N. Towards a discipline of real time programming. Comm. ACM 20, 8 (Aug. 1977), 577-583.

Wod72a Wodon, P. Still another tool for controlling cooperating algorithms. Carnegie-Mellon Univ. Report.

Wod72b Wodon, P. The Belpaire-Wilmotte method for transforming up/down operations into P/V operations. Unpublished manuscript.

Zav76 Zave, P. On the formal definition of processes. Proc. of the 1976 Conf. on Parallel Processing, Aug. 1976, pp. 35-42.

GLOSSARY AND DEFINITION INDEX


LME